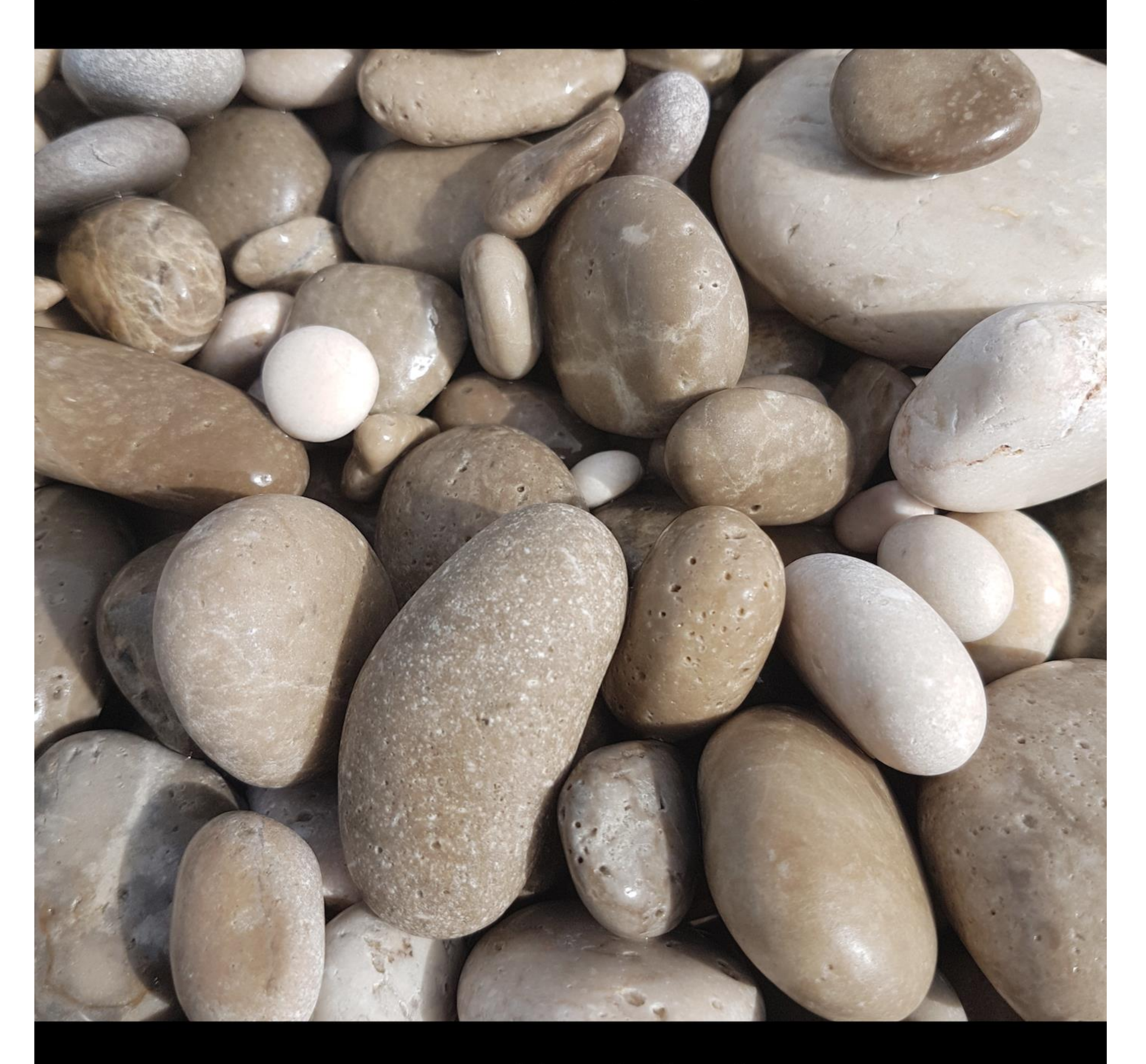
ستيف ماكونيل

الشفرة الكاملة

الإصدار الثاني

مرجع عملي في بناء التطبيقات



ترجمة على هلال عيسى

إشراف ومراجعة زايد السعيدي

الشفرة الكاملة

الإصدار الثاني

تأليف:

ستيف ماكونيل

الإعداد والإشراف والمراجعة للنسخة العربية:

زايد السعيدي

ترجمة:

علي هلال عيسى

بدعم من:

وادي التقنية

www.itwadi.com

نسخة 1.0

1440

عن الكتاب

أصل هذا العمل ترجمة متصرفة لكتاب Code Complete- second edition، تأليف ستيف ماكونيل، قام بالترجمة العربية موقع وادي التقنية وإشراف عام ومراجعة **زايد السعيدي**.





إذا صادفت أي من الأخطاء الإملائية واللغوية أو في الترجمة يرجى مراسلتنا عبر هذه الصفحة:

https://itwadi.com/contact

عن المؤلف

ستيفن ماكونيل، مؤلف كتاب الشفرة الكاملة والعديد من كتب هندسة البرمجيات الأخرى مثل التطوير السريع "Rapid Development "، وتقييم البرمجيات "Software Estimation". ويشار إليه كخبير في هندسة البرمجيات وإدارة المشاريع.

حصل ماكونيل على درجة البكالوريوس في علوم الحاسب في كلية ويتمان، واشنطن، ودرجة الماجستير في هندسة البرمجيات من جامعة سياتل. ثم عمل بعد ذلك في مجال صناعة البرمجيات المكتبية، حيث عمل في مايكروسوفت، وبوينغ، ومجموعة راسل للاستثمار والعديد من شركات منطقة سياتل الأخرى.

نشر ماكونيل كتابه الأول، الشفرة الكاملة، في عام 1993.

من عام 1996 إلى 1998، كان رئيس تحرير خانة "أفضل الممارسات" في مجلة IEEE Software. ومن 1998 إلى 2002، شغل منصب رئيس تحرير المجلة.

عن وادي التقنية

وادي التقنية موقع تقني عربي يُعنى بتتبع أخبار البرمجيات الحرة والمواد التعليمية المتعلقة بها، يكتب فيه عدد من المتطوعين المهتمين بالبرمجيات الحرة والتقنية بشكل عام، يهتم وادي التقنية بمواضيع مثل أنظمة التشغيل الحاسوبية والهاتفية، ولغات البرمجة، والمكتبات البرمجية، وتقنيات الويب، وأخبار شركات البرمجة الكبرى، والمصادر المفتوحة، والعتاد، وأجهزة الحاسوب والهواتف.

قام وادي التقنية بدعم كتابة العديد من الكتب التقنية في مجال البرمجيات الحرة ومفتوحة المصدر، وتوفيرها مجانا للمستخدم التقني العربي، من أهم الكتب التي دعمها وادي التقنية:

تعلم جافا سكربت، دفتر مدير دبيان، سطر أوامر لينُكس، انطلق في انكسكيب، تعرف على البرمجيات الحرة، بوستجريسكل كتاب الوصفات.

لزيارة وادي التقنية وتقديم الدعم له من هنا:

http://itwadi.com/

مقدمة لا بد لها:

في يوم من الأيام كنت في رحلة عمل إلى الصين، وكانت الشركة المستضيفة شركة تقنية هندسية، وكان برفقتنا مترجم، نعم اللغة الإنجليزية لغة عالمية، ولكن تبدأ حدودها عندما تخرج من الصين، أما في داخلها فلا أحد يجيدها، وكل التعليم والدراسة والبحث والتطوير يتم باللغة الصينية.

والحال نفسه في اليابان وأغلب الأمم المتقدمة، اللغة الأم هي الأساس واللغة الإنجليزية هي المكمّل، لذا ترى حركة الترجمة لا تهدأ، فشخص يترجم وعشرات الألوف يركزون على المعلومة بدون تشتيت الجهد في عملية الترجمة لكل شخص.

هذا حال الأمم المتقدمة، وكذاك الحال في الأمم التي تريد التقدم، فقد كنت في زيارة لألبانيا، وهي دولة في شرق أوربا لا يربو تعدادها عن بضعة ملايين، ولكن كل شيء باللغة الألبانية، وقد سألت مضيفي هل يواجهون مشكلة في الترجمة مع قلة الأشخاص، فأكّد أنهم لا يعانون من هذه المشكلة!!

ونحن أمة مئات الملايين، نواجه أزمة في الترجمة، العيب ليس في اللغة بل في أنفسنا.

جلد الذات لن يغير الحال، لذا قررنا أن نساهم بما نستطيع، وأن نترجم ما نحب من العلوم، والحمد لله أخرجنا عدّة كتب، وهذا الكتاب "الشفرة الكاملة" يعتبر من أضخم الأعمال التي نقلناها للغة العربية جهدا ومالا، فهو من المصادر الرفيعة في تقنيات تطوير البرمجيات بغض النظر عن اللغات أو الأطر المستخدمة عملية البرمجة بذاتها.

الكتاب يعلمك كيف تصمم تطبيق ليدوم، مستخدما ذخيرة واسعة من التجارب والأبحاث والكتب في هذا المجال.

أتمنى أن يكون هذا الكتاب في يدِ كل أستاذٍ جامعي وطالبٍ ومطوّرٍ وهاوٍ في هذا المجال لتعمّ الفائدة، لذا جعلته تحت رخصة مفتوحة ليتمكن الجميع من الاستفادة منه.

إذا كنت صاحب دار نشر، فيسعدنا التعاون لإخراج نسخة مطبوعة.

هذا العمل لم يكن ليرى النور لولا تفاني المترجم على هلال عيسى، والذي ساعدتنا خلفيته التعليمية - حيث أنه درس التقنية باللغة العربية - في تسهيل العديد من العقبات، التي نواجهها مع المترجمين الجدد.

وأخيرا أسأل الله تعالى أن يجعل هذا العمل طهارة لمالنا وعلمنا وأن ينفع به المسلمين.

زايد السعيدي

مسقط

25 صفر 1440هـ

3 نوفمبر 2018م

فهرس المحتويات



تمهيد

إنّ الفجوة بين أنشطة هندسة البرمجيات المميّزة والمعدل الوسطي للأنشطة البرمجية واسعة جداً – وقد تكون أوسع من أيّ من تخصصات الهندسة الأخرى. "الأداة التي تنشر أفكار لأنشطة عملية جيدة لابد أن تكون مهمة" ***فريد- بروكس.***

هدفي الرئيسي من كتابة هذا الكتاب هو تضييق الفجوة بين المعرفة الموجودة لدى معلمي هذه الصناعة والأساتذة من ناحية، والأنشطة العملية التجارية الشائعة من ناحية أخرى. حيث أن العديد من تقنيات البرمجة القوية تبقى مخبأة بين طيّات المجلات والأوراق الأكاديمية لسنوات قبل وصولها إلى الاستخدام الفعلي في البرمجة العامة.

وعلى الرغم من أن الأنشطة الرائدة في مجال تطوير البرمجيات قد تقدمت بسرعة في السنوات الأخيرة، إلاّ أنّ الأنشطة الشائعة منها ليست كذلك. فالعديد من البرامج سريعة العطب، وبطيئة، ومكلفة فوق الحد المقبول، والعديد منها تفشل في تلبية احتياجات مستخدميها.

اكتشف الباحثون في كل من صناعة البرمجيات والمعدّين الأكاديميين الممارسات الفعالة التي تحل معظم المشاكل البرمجية التي كانت سائدة منذ السبعينيات.

إنّ هذه الأنشطة لا يتم الإبلاغ عنها في كثير من الأحيان خارج صفحات المجلات التقنية المتخصصة جداً. ومع ذلك فإن معظم منظمات البرمجة لم تستخدمها حتى اليوم. وقد وجدت الدراسات أن تطوير البحوث عادةً ما يستغرق من 5 إلى 15 سنة أو أكثر حتى تأخذ طريقها إلى الممارسة التجارية.

(راغافان وتشاند Raghavan and Chand 1989، روجرز Rogers 1995، بارنز Parnas 1999).

هذا الدليل يختصر هذه العملية، جاعلاً الاكتشافات الرئيسية متاحة الآن للمبرمج المتوسط.

من عليه أن يقرأ هذا الكتاب؟

إن هذا البحث والخبرة البرمجية التي تم جمعها في هذا الدليل ستساعدك على إنشاء برامج ذات جودة أعلى والقيام بعملك بسرعة أكبر وبمشاكل أقل.

حيث سيعطيك الكتاب فكرة عن سبب حدوث المشكلات في السابق ويبين لك كيفية تجنب المشاكل في المستقبل.

كما أن تمارين البرمجة المعروضة هنا سوف تساعدك على الإبقاء على المشاريع الكبرى تحت السيطرة، كما ستساعدك في الصيانة والتعديل على البرمجيات بنجاح مع تغير متطلبات مشاريعك.

المبرمجون الخبراء

يخدم هذا الدليل أيضاً المبرمجين ذوي الخبرة الذين يريدون دليلاً شاملاً وسهلاً للاستخدام لتطوير البرمجيات. لأن هذا الكتاب يركز في بناءه على الجزء الأكثر ألفة من دورة حياة البرنامج. فهو يجعل تقنيات تطوير البرمجيات القوية مفهومة للمبرمجين المتدربين بشكل ذاتي، كما هو الحال بالنسبة للمبرمجين الذين تلقوا تدريباً رسمياً.

الإرشادات التقنية

استخدم كتاب " الشفرة الكاملة " العديد من الإرشادات التقنية لتثقيف المبرمجين الأقل خبرة ضمن فرقهم. ويمكنك أيضاً استخدامه لملء الثغرات المعرفية لديك.

إذا كنت مبرمجاً خبيراً، فقد لا تتفق مع كل استنتاجاتي - وسأكون متفاجئاً إن فعلت- ولكن إذا قرأت هذا الكتاب وفكرت في كل مسألة، فإنه نادراً ما سيطرح عليك أحدهم قضية بناء برمجية لم تعرفها مسبقاً.

المبرمجون ذاتيو التعلم

إذا لم تتلقى الكثير من التدريب الرسمي، وكنت في شركة جيدة. فإن حوالي /50000/ مطور جديد يدخلون المهنة كل عام (BLS 2004، Hecker 2004)، ولكن تُمنح سنوياً حوالي /35000/ شهادة فقط متعلقة بالبرامج (NCES 2002). هذا ما يستدعي إلى الاستنتاج بأن العديد من المبرمجين لم يتلقوا تعليماً رسمياً في تطوير البرمجيات. يتواجد المبرمجون ذاتيو التعلم في مجموعات منشأة من قبل مهندسين احترافيين، ومحاسبين، وعلماء، ومدرسين، وأصحاب الأعمال الصغيرة الذين يبرمجون كجزء من عملهم لكنهم لا يُظهرون أنفسهم بالضرورة كمبرمجين. بغض النظر عن مدى ثقافتك البرمجية، هذا الدليل يمكنه أن يقدّم لك فكرة عن أنشطة البرمجة الفعّالة.

الطلاب

إنّ المتخرج حديثاً يكافئ مبرمج لديه خبرة لكن مع القليل من التدريب الرسمي. فالمتخرجون الجدد في الغالب أغنياء بالمعارف النظرية ولكنهم في الوقت نفسه يفتقرون إلى المعرفة العملية التي تلزمهم لبناء وإنتاج البرامج.

وغالباً ما يتم نقل المعرفة العملية لكتابة الشفرة الجيدة (Coding) ببطء في الأوساط التقليدية القديمة من مهندسي بناء البرمجيات، ومديري المشاريع، والمحللين، والمبرمجين الأكثر خبرة. حتى في كثير من الأحيان، تكون هذه المعرفة نتاج تجارب وأخطاء المبرمج الشخصية.

هذا الكتاب هو البديل عن العمل البطيء للفكر التقليدي، حيث يجمع كلاً من النصائح المفيدة واستراتيجيات

التطوير الفعالة المتاحة مسبقاً بشكل رئيسي عن طريق التقاط وتجميع خبرة الآخرين. وسيكون بمثابة اليد التي تنقل الطالب من البيئة الأكاديمية إلى البيئة الاحترافية.

أين يمكنك أيضاً العثور على هذه المعلومات؟

يجمع الكتاب تقنيات البناء من مجموعة متنوعة من المصادر المنتشرة على نطاق واسع، والكثير من الخبرة المتراكمة حول بناء البرمجيات التي بقيت كمصادر خارجية غير مكتوبة لسنوات. (هيلدبراند 1989، مكونيل 1997). ليس هنالك شك حول فعالية تقنيات البرمجة القويّة المستخدمة من قبل المبرمجين الخبراء في الاندفاع قدماً في المشاريع الحديثة يوماً بعد يوم. ومع ذلك فإن عدداً قليلاً من الخبراء يأخذون الوقت الكافي ليشاركوا ما تعلموه. وبالتالي، فقد يجد المبرمجون صعوبةً في العثور على مصدر جيد للمعلومات البرمجية.

التقنيات الموضحة والمشروحة في هذا الكتاب تقوم بملء الفراغ الذي ستجده بين النصوص التمهيدية ونصوص البرمجة المتقدمة.

ما هو الكتاب الذي تقرأه لمعرفة المزيد عن البرمجة، بعد أن تنتهي من قراءة مقدمة إلى جافا، جافا المتقدمة، وجافا المتقدمة المعمّقة؟ يمكنك قراءة كتب حول تفاصيل أجهزة إنتل أو موتورولا، نظام تشغيل مايكروسوفت ويندوز أو وظائف نظام التشغيل لينوكس، أو لغة برمجة أخرى – لا يمكنك استخدام لغة أو برنامج في بيئة ما بدون الإشارة جيداً إلى مثل هذه التفاصيل. ولكن هذا الكتاب واحد من الكتب القليلة التي تناقش البرمجة في حد ذاتها.

بعض من أكثر الوسائل المساعدة فائدةً هي الأنشطة التي يمكنك القيام بها بغض النظر عن البيئة أو اللغة التي تعمل فيها. الكتب الأخرى تهمل عموماً هذه الممارسات. وهذا ما حدا بنا إلى التركيز عليها في هذا الكتاب.

المعلومات الموجودة في هذا الكتاب مأخوذة من مصادر عديدة، كما هو مبين أدناه.
الطريقة الأخرى الوحيدة للحصول على المعلومات التي ستجدها في هذا الدليل هي البحث في جبل من الكتب وبضع مئات من المجلات التقنية وبعد ذلك إضافة كمية كبيرة من الخبرة العملية الحقيقة.

إذا كنت قد قمت بكل ذلك بالفعل، فإنه لا يزال بإمكانك الاستفادة من هذا الكتاب الذي يجمع المعلومات في مكان واحد كمرجع سهل.



الفوائد الرئيسية لهذا الدليل:

مهما كانت خلفيتك التقنية، يمكن أن يساعدك هذا الدليل على كتابة برامج أفضل في وقت أقل ومع كمية قليلة من المتاعب.

مرجع بناء البرمجيات الكامل: يناقش هذا الدليل الجوانب العامة من البناء مثل جودة البرمجيات وطرق التفكير في البرمجة. كما أنه يصل الى تفاصيل البناء الأساسية مثل الخطوات في بناء الصفوف (classes)، ودخل وخرج البيانات المستخدمة، وهياكل التحكم، والتصحيح، وإعادة الهيكلة، وتقنيات واستراتيجيات ضبط الشفرة. لن تكون بحاجة لقراءة الكتاب من الغلاف إلى الغلاف لمعرفة المزيد عن هذه المواضيع. فقد صُمم الكتاب بحيث يصبح من السهل العثور على المعلومات المحددة التي تهمك.

قوائم اختبار جاهزة للاستخدام: يتضمن هذا الكتاب العشرات من قوائم الاختبار التي يمكنك استخدامها لتحديد هيكلية البرمجيات الخاصة بك، ومنهج التصميم، وجودة الصفوف والإجرائيات، وأسماء المتغيرات، وهياكل التحكم، والتخطيط، وحالات الاختبار، وأكثر من ذلك بكثير.

أفضل المعلومات: يصف هذا الدليل بعضاً من أحدث ما توصلت إليه التقنيات المتاحة، والكثير منها لم تصبح ضمن الاستخدام الشائع حتى الآن. ولأن هذا الكتاب يجمع بين الممارسة العملية والأبحاث، فإن التقنيات التي يصفها ستظل مفيدةً لسنوات.

منظور أوسع لتطوير البرمجيات: سيعطيك هذا الكتاب فرصة للترفع عن الجدال والمشاكل البرمجية اليومية ومعرفة ما هو الشيء القابل للتطبيق أو غير القابل للتطبيق. عدد قليل من المبرمجين المتمرسين لديهم الوقت لقراءة المئات من الكتب ومقالات المجلات التي تم تضمينها في هذا الدليل.

إنّ البحث العلمي والخبرة الواقعية التي تم جمعها في هذا الكتيب سوف تحفز تفكيرك حول المشاريع الخاصة بك، مما يتيح لك اتخاذ إجراءات استراتيجية بحيث لن تضطر للوقوع في نفس الأخطاء مرة تلو الأخرى.

التركيز: نسبة المعرفة في بعض كتب البرمجيات الى الكلام غير المفيد مساوية لـ (10/1). يقدم هذا الكتاب مناقشات متوازنة حول نقاط القوة والضعف في كل تقنية. فأنت تعرف مطالب مشروعك الخاص أفضل من أي شخص آخر. كما يقدم لك هذا الكتاب أيضاً المعلومات الموضوعية التي تحتاجها لاتخاذ قرارات جيدة تبعاً لظروفك.

مفاهيم تنطبق على معظم اللغات الشائعة: يصف هذا الكتاب التقنيات التي يمكنك استخدامها للحصول على أقصى استفادة من أي لغة تستخدمها، سواء كانت سي ++ (C ++) أو سي شارب (C #) أو جافا (Java) أو ميكروسوفت فيجوال بيسك (Microsoft Visual Basic) أو أية لغات أخرى مشابهة.

العديد من أمثلة الشيفرات البرمجية: يحتوي الكتاب على ما يقرب من 500 مثال من الشفرات الجيدة وغير الجيدة. لقد أوردت الكثير من الأمثلة لأنني شخصيّاً، أتعلم بشكل أفضل منها. وأعتقد أن المبرمجين الآخرين كذلك يتعلمون بشكل أفضل بهذه الطريقة.

الأمثلة هي من لغات متعددة لأن اتقان أكثر من لغة واحدة غالباً ما يكون نقطة تحول في مسيرة المبرمج المحترف. بمجرد أن يدرك المبرمج أن مبادئ البرمجة تتجاوز القواعد النحوية للغة معينة، ستفتح له الأبواب للمعرفة التي ستُحدث فرقاً حقيقياً في الجودة والإنتاجية.

لتخفيف عبء تعدد اللغات قدر الإمكان تجنبت الخوض في ميزات اللغة التفصيلية إلا تلك التي تتم مناقشتها على وجه التحديد. لا تحتاج إلى فهم كل تفصيل بسيط من أجزاء التعليمات البرمجية لفهم النقاط التي تقوم بإنجازها. إذا كنت تركز على هذه النقطة التي يجري توضيحها، ستجد أنه بإمكانك قراءة الشفرة بغض النظر عن اللغة المستخدمة. ولقد حاولت جعل عملك أسهل من خلال التعقيب على أجزاء كبيرة من الأمثلة.

الوصول إلى مصادر المعلومات الأخرى: هذا الكتاب يجمع الكثير من المعلومات المتاحة عن بناء البرمجيات، ولكن هذا ليس كل شيء، فخلال فصول هذا الدليل، ضمن أقسام "موارد إضافية " ستجد وصفاً للكتب والمقالات الأخرى التي يمكنك قراءتها أثناء متابعة المواضيع الأكثر إثارة لاهتمامك.

موقع الكتاب الالكتروني: يوفر قوائم المراجعة المحدثة والكتب والمقالات والمجلات، وروابط الويب، والمحتويات الأخرى على الموقع المرفق في cc2e.com.

للوصول إلى المعلومات المتعلقة بالإصدار الثاني لكتاب "الشفرة الكاملة"، أدخل الى cc2e.com/ متبوعا برمز

مكون من أربعة أرقام، ويرد مثال على ذلك هنا في الهامش. تظهر مراجع مواقع الويب هذه في جميع أنحاء الكتاب[1].

لماذا كتب هذا الدليل

الحاجة إلى كتيبات التطوير التي تؤمّن المعرفة حول ممارسات التطوير الفعالة المعترف بها جيداً في مجتمع هندسة البرمجيات.

فقد ذكر تقرير لمجلس علوم وتكنولوجيا الحاسوب أن أكبر المكاسب في جودة تطوير البرمجيات وإنتاجيتها سوف تأتي من التدوين والتكامل وتوزيع المعارف الموجودة بشأن الممارسات الفعالة لتطوير البرمجيات (CSTB 1990، McConnell 1997a). وخلص المجلس إلى أن استراتيجية انتشار تلك المعرفة ينبغي أن تبنى على مفهوم مراجع هندسة البرمجيات.

موضوع البناء قد تم اهماله

في وقت ما، كان يعتقد أن تطوير البرمجيات وكتابة الشفرة هما شيء واحد. ولكن بعد أن عُرفت أنشطة مختلفة في دورة حياة تطوير البرمجيات، بعض من أفضل العقول في هذا المجال قد قضوا أوقاتهم في تحليل ومناقشة أساليب إدارة المشاريع، والمتطلبات، والتصميم، والاختبار.

أغفل الاندفاع لدراسة هذه المجالات المحددة حديثاً بناء الشفرة بوصفه " ابن عم غير مهم" لتطوير البرمجيات.

وقد تعثرت أيضا المناقشات حول البناء من خلال الاقتراح بأن معالجة البناء كنشاط تطوير برمجيات مستقل، يعني بأن البناء أيضاً يجب أن يعامل كمرحلة مستقلة.

في الواقع، ليس من الضرورة أن ترتبط الانشطة والمراحل المختلفة للبرمجيات بعلاقات مع بعضها، ومن المفيد مناقشة نشاط البناء بغض النظر عما إذا كانت أنشطة البرمجيات الأخرى تُنجز على مراحل، تكرارات، أو بطرق أخرى.

البناء مهم

هناك سبب آخر في أن البناء قد تم إهماله من قبل الباحثين والكتّاب، وهو الفكرة الخاطئة التي تقول: إنه بالمقارنة مع أنشطة تطوير البرمجيات فإن عملية البناء هي عملية فيزيائية تتيح فرصة قليلة للتطوير، وهذا أبعد ما يكون عن الصواب.

[1] cc2e.com/1234

عادةً ما يشكل بناء الشفرة نحو 65 ٪من الجهد المبذول في المشاريع الصغيرة و50 ٪من ذلك الجهد في المشاريع المتوسطة. ويشكل البناء حوالي 75 ٪من الأخطاء في المشاريع الصغيرة ومن 50٪ إلى 75٪ في المشاريع المتوسطة والكبيرة. وأي نشاط يحوي من 50٪ إلى 75٪ من الأخطاء يعتبر فرصة مناسبة للقيام بالتحسين (يحتوي الفصل 27 على مزيد من التفاصيل عن هذه الإحصاءات).

وقد أشار بعض المعلقين إلى أنه وعلى الرغم من أن أخطاء البناء تمثل نسبة عالية من إجمالي الأخطاء، إلا أن أخطاء البناء غالباً ما تكون كلفة لإصلاحها أقل قياساً مع تلك الأخطاء الناجمة عن المتطلبات والبنية، وبالتالي كان الاقتراح بأن يتم اعتبارها أقل أهمية.

إن الادعاء بأن تكلفة أخطاء البناء أقل من الإصلاح صحيح ولكنه مضلل، لأن تكلفة عدم إصلاحها يمكن أن تكون مرتفعة بشكل كبير جداً. وقد وجد الباحثون أن الأخطاء الصغيرة في كتابة الشفرة تمثل بعضاً من أخطاء البرمجيات الأكثر كلفة على مر الأيام، حيث تصل التكاليف إلى مئات الملايين من الدولارات (Weinberg 1983، SEN 1990). إن تكلفة الإصلاح غير المرتفعة لا تعني أن الإصلاح يجب أن يكون منخفض الأولوية.

والمثير للسخرية من تحول التركيز بعيداً عن البناء، هو أن ذلك البناء هو النشاط الوحيد الذي تستطيع ضمان القيام به، حيث يمكن افتراض المتطلبات بدلا من وضعها؛ أسلوب البناء يمكن أن يكون أقل تغيراً من التصميم؛ والاختبار يمكن اختصاره أو تخطيه بدلاً من التخطيط له وتنفيذه بالكامل. ولكن إذا كان هناك برنامج، فلا بد أن يكون هنالك بناء، وهذا ما يجعل البناء مجالاً مثمراً وفريداً لتحسين أنشطة التطوير.

لا يوجد كتاب متاح للمقارنة

وفي ضوء أهمية البناء الواضحة، كنت متأكداً عندما فكّرت في إنجاز هذا الكتاب أن شخصاً آخر قد قام بالفعل بكتابة كتاب عن ممارسات البناء الفعّالة. ويبدو أن هناك حاجة ملحّة إلى كتاب عن كيفية البرمجة الفعّالة.

لكنني وجدت عدداً قليلاً فقط من الكتب كانت قد كتبت عن البناء وتحوي فقط على أجزاء من هذا الموضوع. وقد كُتب بعضها قبل 15 سنة أو أكثر، واستخدموا لغات محدودة نسبياً مثل ALGOL وPL/I وRatfor وSmalltalk. بعض من هذه المواضيع كتبها أساتذة جامعيين لم يعملوا مسبقا على كتابة الشفرات البرمجية. كتب هؤلاء الأساتذة عن التقنيات التي تعمل في مشاريع الطلاب، ولكن في كثير من الأحيان كان لديهم فكرة بسيطة عن كيفية عمل هذه التقنيات في بيئات التطوير واسعة النطاق، ومع ذلك لا تزال الكتب الأخرى تهلل بأفضل المنهجيات المفضلة للمؤلفين، ولكنها تجاهلت بذلك مستودع ضخم من الأنشطة المكتملة التي أثبتت فعاليتها مع مرور الوقت.

وباختصار، لم أتمكن من العثور على أي كتاب قد حاول حتى الاحتواء على مجموعة من التقنيات العملية المأخوذة من الخبرة المهنية، والأبحاث الصناعية، والعمل الأكاديمي.1 يتوجب تحديث وتطوير النقاش في لغات البرمجة الحالية، والبرمجة غرضية التوجه object-oriented programming، وأنشطة التطوير الرائدة. وقد بدا واضحاً الآن الحاجة لكتابة كتاب عن البرمجة من قبل شخص على دراية نظرية وقام أيضاً ببناء ما يكفي من الشفرات البرمجية لتقدير حالة هذه الممارسة المهنية. لقد فكرت في هذا الكتاب كنقاش كامل بين المبرمجين لبناء الشفرة البرمجي.

[1] عندما يجتمع نقاد الفن معا يتحدثون عن النموذج والهيكل والمعنى. وعندما يجتمع الفنانون يتحدثون عن أين يمكن شراء وقود رخيص. -بابلو بيكاسو

ملاحظة المؤلف

أرحب باستفساراتكم حول الموضوعات التي نوقشت في هذا الكتاب، وتقارير الأخطاء، أو أيّة مواضيع أخرى ذات صلة. يرجى الاتصال بي في stevemcc@construx.com، أو زيارة الموقع الإلكتروني: www.stevemcconnell.com

بيليفو، واشنطن
يوم الذكرى، 2004

الدعم التقني التعليمي لمايكروسوفت

لقد بذل كل جهد ممكن لضمان دقة هذا الكتاب. تقدم مايكروسوفت بريس تصحيحات للكتب من خلال الموقع على الشبكة العالمية على العنوان التالي:

/http://www.microsoft.com/learning/support

للاتصال مباشرة إلى "قاعدة المعارف لدى مايكروسوفت" والاستعلام بخصوص سؤال أو مسألة قد تكون لديكم، انتقل إلى:

http://www.microsoft.com/learning/support/search.asp

إذا كانت لديك تعليقات أو أسئلة أو أفكار تتعلق بهذا الكتاب، فيرجى إرسالها إلى مايكروسوفت بريس باستخدام أي من الطرق التالية:

بريد:

Microsoft Press
Attn: Code Complete 2E Editor
One Microsoft Way
Redmond ,WA 98052-6399

البريد الإلكتروني:

mspinput@microsoft.com

اعتراف بالجميل:

الكتاب لا يكتب أبداً من قبل شخص واحد - على الأقل أيّاً من كتبي ليس كذلك- الطبعة الثانية هي أقرب ما تكون إلى مشروع جماعي.

أحبّ أن اشكر الناس الذين أسهموا بتنقيحات في مجالات هامة من الكتاب، وهم: هاكون أغوستسون، سكوت أمبلير، ويل بارنز، ويليام دي، بارثولوميو، لارس بيرغستورم، إيان بروكبانك، بروس بتلر، جاي سنكوتا، الان كوبر، بوب كوريك، الكوروين، جيري ديفل، جون إيفز، إدوارد إسترادا، ستيف غولدستون، أواين غريفثز، ماثيو هاريس، مايكل هوارد، أندي هنت، كيفين هاتشهسون، روب جاسبير، ستيفين جنكنز، رالف جونسون ومجموعته لتصميم البرمجيات في جامعة إيلينويس، ماريك كونوبكا، جف لانغر، أندي ميستر، ميتيكا مانو، ستيف ماتينغلي، غارث مكّان، روبرت مكوفيرن، سكوت مايرز، غارث مورغان، مات بيموكوين، براين بفلوغ، جيفيري ريتشتر، سبيف رن، دوغ روسينبيرغ، براين ست. بيير، ديوميديس سبينلس، مات ستيفينز، ديف توماس، أندي توماس-كرامير، جون فميسيديس، بافل فوزينيمك، دني ويليفورد، جاك وولّي، زسومبور.

أرسل المئات من القراء تعليقات عن الطبعة الأولى، وعدد أكبر من هؤلاء القراء أيضاً أرسلوا تعليقات فردية عن الطبعة الثانية. الشكر لكل من قضى وقتاً يشارك استجابته لهذا الكتاب بشتى أشكالها.

شكر خاص الى نقاد "كونستركس سوفتوير" الذين فحصوا كامل مخطوطة الكتاب بشكل رسمي وهم: جاسون هيلز، برادي هونسينغير، عبدول نزار، توم ريد، باميلا بيروت. وكنت مندهشاً تماماً كم كانت انتقاداتهم عميقة، خصوصاً بالنظر الى عدد الأعين التي أمعنت النظر بالكتاب قبل ان يبدؤوا العمل به. الشكر أيضاً إلى برادي، جاسون، باميلا لتبرعاتهم لموقع cc2e.com.

استمتعت كثيراً بالعمل مع ديفون موسغراف رئيس التحرير لهذا الكتاب. لقد عملت مع العديد من المحررين الممتازين في مشاريع أخرى، وديفون يمثل على وجه الخصوص الوجدانية وسهولة التعامل. شكراً ديفون! الشكر للندا انغلمان التي أيدت الطبعة الثانية؛ هذا الكتاب ما كان ليكون بدونها. الشكر أيضاً لبقية طاقم ""مايكروسوفت برس"، المتضمن: روبن فان ستينبرغ، إلدين نيلسون، كارل ديلتز، دويل بانكوف، باتريشيا ماسرمان، بل مايرز، ساندي ريسنك، باربارا نورفليت، جيمس كارمير، بريسكوت كلاسين.

أحب أن أذكر طاقم "مايكروسوفت برس" الذي أصدر الطبعة الأولى: أليس سمث، أرمين مايرز، باربارا رونيان، كارول لوك، كوني لتل، دين هولمز، إيرك استرو، إرين أوكونور، جيني مكيفرن، جف كاري، جينيفر هاريس،

جينيفر فك، جوديث بلوك، كاترين إيريكسون، كيم إيغلستون، ليزا ساندبرغ، ليزا ثيوبالد، مارغاريت هارغريف، مايك هالفورسن، بات فورغيت، بيغي هيرمان، روث بيتيس، سالي برونزمان، شاون بك، ستيف موري، واليس بولز، جعفر حسنين.

الشكر للنقاد الذين ساهموا بشكل ملحوظ في الطبعة الأولى وهم: الكوروين، بيل كيستلير، براين داوغيري، ديف مور، غريغ هيتشكوك، هانك ميوريت، جاك وولي، جوي ويريك، مارغوت بيغ، مايك كلين، مايك زيفينبيرغين، بات فورمان، بيتير بيث، روبيرت ل. غلاس، تامي فورمان، توني بيسكولي، وأين بيردسمي. شكر خاص لتوني غارلاند لنقده المفصل: بإدراكي لـ 12 سنة مضت، أقدر أكثر من أي وقت كم كانت استثنائية حقاً انتقادات توني العدة آلاف.

القسم الأول: ترتيب الأساسيات

في هذا القسم:

الفصل الأول: مرحباً بك في بناء البرمجيات.

الفصل الثاني: الاستعارات، لفهم أفضل لتطوير البرمجيات.

الفصل الثالث: قس مرتين واقطع مرة، المتطلبات التحضيرية.

الفصل الرابع: قرارات بناء مفتاحية.

الفصل الأول

مرحبا بك في بناء البرمجيات

المحتويات[1]:

مواضيع ذات صلة:

- من تنبغي عليه قراءة هذا الكتاب: المقدمة
- فوائد قراءة هذا الكتاب: المقدمة
- لماذا كتب هذا الكتاب: المقدمة

قد تعلم معنى كلمة "بناء" خارج مجال تطوير البرمجيات. "البناء" هو عمل "عمال البناء" عندما ينشؤون بيتاً أو مدرسةً أو ناطحة سحاب. عندما كنت أصغر سناً قمت بإنشاء أشياء من خلال "ورقة البناء." في الاستخدام الشائع "البناء" يشير إلى عملية الإنشاء. عملية البناء يمكن أن تحوي جوانب من التخطيط والتصميم وتفحص العمل الجاري، لكن غالباً "البناء" تشير إلى الجزء المتعلق بالعمل اليدوي من إنشاء شيء ما.

1.1ما هو بناء البرمجيات؟

تطوير برمجيات الحاسوب يمكن أن تكون عملية معقدة، وفي الـ 25 سنة الماضية، عرّف الباحثون عدة أنشطة مميزة تكّون تطوير البرمجيات. ومنها:

- تعريف المشكلة.
- تطوير المتطلبات.
- التخطيط للبناء.
- هندسة البرمجيات، أو التصميم عالي المستوى.
- التصميم التفصيلي.

[1] cc2e.com/0178

- كتابة الشفرة وتصحيحها.
- اختبار الوحدة.
- اختبار التكامل.
- التكامل.
- اختبار النظام.
- الصيانة التصحيحية.

إذا كنت قد عملت في مشاريع غير رسمية، فربما تعتقد أنّ هذه الشروط تمثل روتين بيروقراطي. وإذا كنت قد عملت في مشـــروع رســـمي فإنك تعرف أنّ هذه الشـــروط تمثل روتين بيروقراطي. من الصـــعب أنّ نميّز بين المشاريع الرسمية جدّاً والمشاريع قليلة الرسمية، وهذا ما نوقش لاحقاً في الكتاب.

إذا كنت قد تعلمت البرمجة بشكل ذاتي أو عملت بشكل رئيسي في مشاريع غير رسمية، لن تفرّق بين الأنشطة العديدة التي تدخل في إنشـــاء منتج برمجي. بشـــكل عقلاني، ربما جمعت كل الأنشـــطة معاً تحت مســـمى "برمجة." إذا عملت في مشـاريع غير رسـمية، النشـاط الأسـاسـي الذي تفكر به عندما تريد إنشـاء برمجيّة هو النشاط الذي يسميه الباحثون "البناء".

هذا المفهوم البديهي عن "البناء" دقيق تماماً، لكن بعيد عن المعنى النظري. وضـــع "البناء" في ســـياقه مع الأنشطة الأخرى سيساعدنا على البقاء مركزين على المهام الصحيحة خلال البناء ويؤكد بشكل مناسب أهمية الأنشطة الأخرى غير البناء. الشكل 1-1 يوضح مكان البناء المتعلق بأنشطة تطوير البرمجيات الأخرى.



الشكل 1-1 أنشطة البناء ظاهرة داخل الدائرة الرمادية. البناء يركز على كتابة الشفرة والتصحيح، لكن يتضمن التصميم التفصيلي واختبار الوحدة واختبار التكامل ونشاطات أخرى



كما يو ضح الشكل، البناء هو على نحو غالب كتابة شفره وتصحيحها، لكن يحوي أيضاً التصميم التفصيلي والتخطيط للبناء واختبار الوحدة والتكامل واختبار التكامل ونشاطات أخرى. لو كان هذا الكتاب عن كل مفاهيم تطوير البرمجيات، لقدم نقاشاً متوازناً عن كل الأنشطة في عملية التطوير. وكون هذا الكتاب عن تقنيات البناء، على أية حال، فإنه يتمحور بشكل حيادي على البناء ويمس فقط المواضيع المتعلقة به. فلو مثلنا هذا الكتاب مثلاً بـ "كلب"، سيسكن في البناء، يهز ذيله فوق التصميم والاختبار، وينبح على الأنشطة الأخرى في التطوير.

البناء يعرف أحياناً بـ "كتابة الشفرة" أو "البرمجة." "كتابة الشفرة" ليست حقاً الكلمة الأفضل لأنها تعني الترجمة الآلية لتصميم موجود مسبقاً إلى لغة الحاسوب؛ البناء ليس على الاطلاق أمر آلي بل يتضمن إبداع حقيقي ومحاكمة. خلال الكتاب، قمت باستخدام "برمجة" بشكل متبادل مع "بناء."

على خلاف الشكل 1-1 الذي يمثل منظور الأرض المسطحة لتطوير البرمجيات، الشكل 1-2 يظهر منظور الأرض الكروية لهذا الكتاب.



الشكل 1-2 هذا الكتاب يركز على كتابة الشفرة والتصحيح، التصميم التفصيلي، التخطيط للبناء، اختبار الوحدة، التكامل، اختبار التكامل، وأنشطة أخرى تقريباً بهذا المقدار

الشكلان 1-1 و1-2 هما منظوران عاليا المستوى لأنشطة البناء، لكن ماذا عن التفاصيل؟ فيما يلي بعض المهام المحددة المتضمنة في البناء:

- التأكد من كون الأساس قد وضع وبالتالي يمكن التقدم في البناء بنجاح.
- تحديد كيفية اختبار الشفرة.
- تصميم وكتابة الصفوف والإجرائيات.
- إنشاء وتسمية المتحولات والثوابت المسماة.
- اختيار بنى التحكم وترتيب كتل التعليمات.
- اختبار الوحدة، اختبار التكامل، تصحيح شفرتك الخاص.
- استعراض التصميم المنخفض المستوى وشفره باقي الفريق وجعلهم يستعرضون ما عندك
- تجميل الشفرة بالاعتناء بدقة بالتنسيق، والتعليق عليه.
- تجميع المكونات البرمجية التي أنشأت كل على حدة.
- ضبط الشفرة لتعمل بسرعة أكبر وتقليل الموارد.

إذا أردت القائمة الكاملة لأنشطة البناء، انظر الى عناوين الفصول في جدول المحتويات.

مع العديد من الأنشطة في العمل بالبناء، ربما تقول. "حسناً، جاك، ما هي الأنشطة التي ليست جزء من البناء؟" وهذا سؤال مُحق. الأنشطة المهمة غير البناء تتضمن: الإدارة، وتطوير المتطلبات، وهندسة البرمجية، وتصميم واجهة المستخدم، واختبار النظام، والصيانة. تؤثر كل من هذه الأنشطة في النجاح النهائي للمشروع مثلها مثل البناء؛ على الأقل نجاح أيّ مشروع والذي يتطلب أكثر من شخص أو اثنين ويستمر أكثر من عدة أسابيع. يمكنك أن تجد كتب جيدة عن كل نشاط، العديد منها ذُكر في أقسام "موارد إضافية" خلال هذا الكتاب وفي الفصل 35، " أين تجد المزيد من المعلومات،" في نهاية الكتاب.

1.2 لماذا بناء البرمجية مهم؟

حيث أنك تقرأ هذا الكتاب، ربما توافق أنّ تحسين جودة البرمجية وإنتاجية المطوّر مهمّان، تستخدم الكثير من المشاريع المدهشة في هذه الأيام البرمجيات على نطاق واسع. الانترنت، والتأثيرات السينمائية، والأنظمة الطبية الداعمة للحياة، وعلم الطيران، والتحليل المالي عالي السرعة، والبحوث العلمية أمثلة قليلة. هذه المشاريع ومشاريع معروفة كلها يمكن أن تستفيد من الخبرة الأجود لأن العديد من الأساسات مشتركة.

إذا كنت توافق أنّ تحسين تطوير البرمجيات مهم بالعموم، السؤال لك كقارئ هو: لماذا البناء هو مرتكز هام؟

إليك السبب:

البناء هو قسم كبير من تطوير البرمجية [1] بالاعتماد على حجم المشروع، البناء يأخذ بشكل قياسي من 30 الى 80 بالمئة من الزمن الكلي المصروف للمشروع، أي شيء يستولي على هذا المقدار من وقت المشروع يكون ملتزماً بالتأثير على نجاح المشروع.

البناء هو النشاط المركزي في تطوير البرمجية المتطلبات والهندسة تتمان قبل البناء وبذلك يمكنك أن تقوم بالبناء بشكل فعّال. اختبار النظام (بالمفهوم المقيد للفحص المستقل) يتم بعد البناء ليؤكد أنّ البناء تمّ بشكل صحيح. البناء كائن في مركز عملية تطوير البرمجيات.

بالتركيز على البناء، إنتاجية المبرمج الواحد يمكن أن تتحسن بشكل هائل[2] أظهرت دراسة تقليدية قام بها ساكمان، إريكسون، غرانت أنّ إنتاجية المبرمجين كل على حدة تغيرت بعامل من 10 الى 20 اثناء البناء (1968). منذ وقت دراستهم، أكدت نتائجهم بالعديد من دراسات غيرهم (كورتيس 1981، ميلز 1983، كورتيس إيت أل. 1986، كارد 1987، فاليت وماكاري 1989، دكماركو وليستر 1999، بويم إيت أل. 2000). هذا الكتاب يساعد كل المبرمجين بتعلم تقنيات استخدمت مسبقاً من قبل أفضل المبرمجين.

ناتج البناء، الشفرة المصدرية، غالباً لوحده الوصف الدقيق للبرمجية في العديد من المشاريع، التوثيق الوحيد المتاح للمبرمجين هو الشفرة نفسها. مواصفات المتطلبات ووثائق التصميم يمكن أن تصبح منتهية الصلاحية، لكن الشفرة المصدرية دائماً ضمن الصلاحية. بناء على ذلك، إنه لأمر إلزامي أن تكون الشفرة المصدري بأعلى جودة ممكنة. تطبيقات ملائمة لتقنيات تحسين الشفرة المصدرية تجعل الفرق بين أداة روب غولدبيرغ الغريبة والبرنامج المفصّل، والصحيح، والإعلامي. تطبّق مثل هذه التقنيات بشكل فعال جداً خلال البناء.

[1] **إشارة مرجعية** لتفاصيل حول العلاقة بين حجم المشروع والنسبة المئوية من اوقت المستهلك في البناء، انظر "حصص الأنشطة وحجومها" في القسم 27.5.

[2] **إشارة مرجعية** لمعلومات عن الفوارق بين المبرمجين انظر "الفوارق الفردية" في القسم 28.5.



البناء هو النشـــاط الوحيد الذي يتم ضـــمان تنفيذه المشــروع البرمجي المثالي يبدأ من خلال تحديد المتطلبات بعناية وتصــميم الهيكلية قبل البدء بالبناء الفعلي. ويتطلب المشــروع المثالي اختبارات احصائية شاملة للنظام بعد البناء.

على أي حال، المشاريع الفعلية غير الاحترافية غالباً ما تتجاوز المتطلبات والتصميم وتنتقل الى البناء مباشرة، في هذه المشاريع يقومون بإهمال الاختبارات بسبب وجود العديد من الأخطاء التي يتوجب اصلاحها وبسبب نفاذ الوقت، دون الاكترات بسرعة وسوء تخطيط المشروع.

مشاريع العالم الذي نعيشه الناقصة – على أية حال- تتجاوز المتطلبات والتصميم وتقفز إلى البناء. ويسقطون الاختبار لأن لديهم العديد من الأخطاء ليتم إ صلاحها وقد نفذ منهم الوقت. لكن لا يهم كم المشروع مستعجل أو خُطط له برداءة، لا يمكنك هدم البناء، إنه حيث تلامس العجلة الطريق. تحســين عملية البناء هو طريقة لتحسين أي جهد في تطوير البرمجيات، بغض النظر عن كمية الاختزال.

1.3 كيف تقرأ هذا الكتاب؟

هذا الكتاب مصمم ليُقرأ من الدفة إلى الدفة أو حسب الموضوع. إذا كنت تحبّ أن تقرأ الكتاب من الدفة إلى الدفة، بإمكانك أن تنتقل ببساطة إلى الفصل 2، "استعارات لفهم أفضل لتطوير البرمجيات." إذا كنت تريد الوصول إلى نصائح برمجية محددة، يمكنك أن تبدأ من الفصل 6، "صفوف العمل،" ثم تتبع توجيهات إلى مواضيع تراها ممتعة. إذا كنت محتاراً بين الأمرين، ابدأ بالقسم 3.2، "اختيار نوع البرمجية التي تعمل عليها"

نقاط مفتاحية

- بناء البرمجية هو النشاط الرئيسي في تطوير البرمجيات؛ البناء هو النشاط الوحيد المضمون أن ينفذ في كل مشروع.
- الأنشطة الرئيسية في كل بناء هي: التصميم التفصيلي، وكتابة الشفرة، والتصحيح، والتكامل، واختبار المطور (اختبار الوحدة واختبار التكامل).
- "كتابة الشفرة" و"البرمجة" مصطلحين شائعين للدلالة على البناء.
- جودة البناء تؤثر جوهرياً على جودة البرمجية.
- في التحليل النهائي، فهمك لكيفية القيام بالبناء يحدد كم أنت مبرمج جيد، وهذا موضــوع باقي الكتاب.

الفصل الثاني:

استعارات لفهم أفضل لتطوير البرمجيات

المحتويات

موضوع ذو صلة

- الاستدلال في التصميم: " التصميم هو عملية استدلالية " في القسم 5.1.

يمتلك علم الحاسوب بعضاً من أكثر اللغات تنوعاً ً من أي مجال. في مجال ما آخر يمكنك المشي لغرفة معقمة مضـــبوطة بعناية على الدرجة 68 درجة فهرنهايت، ومع ذلك يمكن أن تجد الفيروســـات، وأحصـــنة طروادة، والديدان، والجراثيم، والقنابل، والحوادث، ومتغيرات الجنس الملتوية، والأخطاء القاتلة.

تصف هذه الاستعارات البيانية الظواهر البرمجية المحددة. وبشكل مماثل تصف الاستعارات الحية ظواهر واسعة النطاق، ويمكن أن تستخدمها لتحسين فهمك في عملية تطوير البرمجيات.

لا تعتمد بقية الكتاب بشكل مباشر على مناقشة الاستعارات في هذا الفصل. بإمكانك أن تتخطاها إن أردت الحصول على الاقتراحات العملية. أو اقرأها إن رغبت بالتفكير في تطوير البرمجيات بشكل أكثر وضوحا ً.

2 . 1 أهمية الاستعارات

غالباً ما تنشأ تطورات هامة من المقارنة. فبمقارنة موضوع تفهمه قليلاً مع شيء ما مشابه له ستفهمه بشكل أفضل، وسيمكّنك من الخروج برؤية تعطي فهم أفضل للموضوع الأساسي. هذا الاستخدام للاستعارة يسمى " النمذجة ".

إن تاريخ العلم مليء بالاكتشافات المبنية على الاستفادة من قوة الاستعارات. كان لدى الكيميائي كيكولي (Kekulé) حلم رأى به أفعى تقبض على ذيلها بفمها. عندما استيقظ، أدرك أن بنية الجزيئات مبنية على شكل

حلقة مشابهه سيكون لها دور في خصائص البنزين. بالإضافة لذلك أكدت التجارب الفرضيات (بربور Barbour 1966).

بنيت النظرية الحركية للغازات على نموذج "كرة البلياردو ". واعتُقِد أن لدى جزيئات الغاز كتلة وتصـــطدم بشكل مطاطي، كما تفعل كرة البلياردو، وطوّرت العديد من النظريات وفقا لهذا النموذج.

طوّرت نظرية موجة الضوء بشكل كبير عن طريق ا ستكشاف التشابهات بين الضوء والصوت. يملك الصوت والضوء السعة (السطوع، شدة الصوت)، التردد (اللون، طبقة الصوت)، وخصائص أخرى مشتركة.

المقارنة بين نظريات الموجة للصوت والضوء كان مثمراً جداً[1]، حيث بذل العلماء قدراً كبيراً من الجهد بالبحث عن الوسيلة التي من شأنها تسهيل الطريق عبر الهواء ليزيد من انتشار الصوت. حتى أنهم أعطوها أسم – " الأثير "- لكنهم لم يجدوا أبداً الوسيلة. أثبت التشابه الذي كان مثمراً جداً في بعض النواحي أنه مضلل في هذه الحالة.

في العموم، تكمن قوة النماذج في أنها واضحة ويمكن أن تمنح مفهوماً كلياً. حيث أنها تقترح الخصائص، والعلاقات، ومجالات إضافية من الاستعلامات. أحياناً يقترح النموذج مجالات من الاستعلام تكون مضللة، في هذه الحالة، يكون التشبيه مبالغاً به. عندما بحث العلماء عن الأثير، تجاوزوا إمكانات نموذجهم.

كما قد تتوقع، تكون بعض التشبيهات أفضل من غيرها. التشبيه الجيد يكون بسيط ويرتبط بشكل جيد مع تشبيهات أخرى ذات صلة ويشرح الكثير من الأدلة التجريبية وظواهر واضحة أخرى.

بالأخذ بعين الاعتبار مثال الحجر الثقيل المتأرجح ذهابا وإيابا على سلسلة. قبل غاليلو (Galileo)، بحث أتباع أرسطو (Aristotelian) في الحجر المتأرجح معتقدين أن الأشياء الثقيلة تتحرك بشكل طبيعي من الموضع الأعلى إلى حالة الاستقرار في الموضع الأدنى. قد يعتقد أتباع أرسطو أن ما كان يفعله الحجر حقيقةً كان السقوط بصعوبة. عندما رأى غاليلو الحجر المتأرجح، رأى النواس، واعتقد أن ما كان يفعله الحجر حقيقة كان إعادة نفس الحركة مراراً وتكراراً، بشكل مثالي تقريباً.

القدرات الموحية للنموذجين مختلفة تماماً. تابع أرسطو الذي رأى الحجر المتأرجح كجسم يسقط سوف يراعي وزن الحجر، والارتفاع الذي وصل اليه، والوقت الذي استغرقه ليستقر. بالنسبة لنموذج النواس لغاليلو، العوامل

[1] يجب ألا يستهان بقيمة الاستعارات. للاستعارات ميزة التصرف المتوقع والمفهوم للجميع. يقلل الاتصالات غير الضرورية وسوء الفهم. تسرع التعلم والتثقيف. في الواقع، الاستعارات هي الوسيلة لفهم وتجريد المفاهيم، تسمح للشخص بالتفكير والتخطيط بشكل أفضل ومستوى أقل من الأخطاء التي يمكن تجنبها. (Fernando J. Corbató)

البارزة كانت مختلفة. راعى غاليلو وزن الحجر، ونصف قطر تأرجح النواس، والانزياح الزاوي، والوقت لكل أرجحة.

اكتشف غاليلو القوانين التي لم يتمكن أرسطو من اكتشافها لأن نموذجهم قادهم للبحث عن ظواهر مختلفة وطرح أسئلة مختلفة.

تساهم الاستعارات في فهم أكبر لقضايا تطوير البرمجيات بنفس الطريقة التي تساهم فيها في فهم أكبر للمسائل العملية. في محاضرته لجائزة تورينغ(Turing) لعام 1973، وصف تشارلز باشمان(Charles Bachman) التغير في الكون من وجهة النظر السائدة المرتكزة على الأرض إلى وجهة النظر المرتكزة على الشمس. لقد استمر نموذج بطليموس(Ptolemy) المرتكز على الأرض دون اعتراض لـ 1400 سنة. ثم في عام 1543، قدم كوبرنيكوس (Copernicus) نظرية مركزية الشمس، فكرة أن الشمس كانت مركز الكون بدلاً من الأرض. أدى هذا التغير في النماذج الذهنية في نهاية المطاف إلى اكتشاف كواكب جديدة، وإعادة تصنيف القمر كتابع بدلا من كوكب، ولفهم مختلف لمكان البشرية في الكون.

قارن باشمان Bachman التغير في علم الفلك من البطلمية (Ptolemaic) إلى الكوبرنيكية (Copernican) بالتغير في برمجة الحاسوب في أوائل السبعينيات. عندما وضع باشمان المقارنة في عام 1973، كانت معالجة البيانات تتغير من وجهة النظر المرتكزة على الحاسوب computer-centered لأنظمة المعلومات إلى وجهة النظر المرتكزة على قاعدة البيانات database-centered. أشار باشمان (Bachman) إلى أن القدماء في معالجة البيانات أرادوا أن يظهروا كل البيانات كتيار متسلسل من البطاقات المتدفقة خلال الحاسوب (وجهة النظر المرتكزة على الحاسوب). كان التغير من أجل التركيز على كمية البيانات التي يتوجب على الحاسب معالجتها (وجهة النظر المرتكزة على قاعدة البيانات).

من الصعب اليوم تخيّل أي شخص يفكّر في أن الشمس تدور حول الأرض. بشكل مشابه، من الصعب تخيل مبرمج يفكّر في أنّه من الممكن أن تظهر كل البيانات كتيار متسلسل من البطاقات. في كلا الحالتين، حالما تستبعد النظرية القديمة، يبدو من غير المعقول على الإطلاق أن شخصاً ما يؤمن بها. من الغرابة أكثر، أن اعتقاد النّاس الذين آمنوا بالنظرية القديمة بأن النظرية الجديدة كانت سخيفة تماماً كما تفكر أنت الآن بالنظرية القديمة.

وجهة النظر للكون المرتكزة على الأرض أعاقت علماء الفلك الذين تشبثوا بها بعدما أتيحت لهم نظرية أفضل. بشكل مشابه، وجهة نظر عالم الحوسبة المرتكزة على الحاسوب أعاقت علماء الحاسوب الذين تمسكوا بها بعدما كانت النظرية المرتكزة على قاعدة البيانات متاحة.

من المغري التقليل من شأن الاستعارات. لكل من الأمثلة السابقة، الردّ الطبيعي هو أن تقول "حسنا، بالطبع إن الاستعارة الصحيحة أكثر إفادة. الاستعارة الأخرى كانت خاطئة! " على الرغم من أن هذا رد فعل طبيعي، إنه تبسيط. تاريخ العلم ليس سلسلة مفاتيح من الاستعارات " الخاطئة " إلى الاستعارات " الصحيحة ". هو سلسلة من التغيرات من الاستعارات " الأسوأ " إلى الاستعارات " الأفضل "، من الأقل شمولية إلى الأكثر شمولية، من الاستعارات الموحية في مجال ما إلى الموحية في مجال آخر.

في الحقيقة، قد استبدلت نماذج عديدة بنماذج أفضل وهي ما تزال مفيدة. ما يزال المهندسون يحلون معظم مشاكل الهندسة باستخدام ديناميكيات نيوتن على الرغم، من الناحية النظرية، أن ديناميكيات نيوتن قد استبدلت بنظرية أينشتاين.

إن تطوير البرمجيات هو الحقل الأحدث من معظم العلوم الأخرى. لكنه ليس ناضج بعد بما يكفي ليملك مجموعة من الاستعارات القياسية. وبالتالي فإن لديه وفرة في الاستعارات التكميلية والمتضاربة. بعضها أفضل من الآخر وبعضها أسوأ. على قدر ما يكون فهمك للاستعارات المحددة جيداً، سيكون فهمك جيداً في تطوير البرمجيات.

2 .2 كيفية استخدام استعارات البرمجيات



استعارة البرمجيات شبيهة بضوء الكشاف أكثر من خارطة الطريق. لا تخبرك أين تجد الحل، بل تخبرك كيف تبحث عنه. تخدم الاستعارة أكثر كونها كاشفة أكثر من كونها خوارزمية.

الخوارزمية هي مجموعة من التعليمات المحددة جيداً لتنفيذ مهمة معينة. ويمكن التنبؤ بالخوارزمية وهي حتمية، وليست عرضه للصدفة. تخبرك الخوارزمية كيف تذهب من نقطة (أ) إلى النقطة (ب) بدون انعطافات. لا يوجد تنقلات جانبية إلى النقط (ج، د، هـ) ودون توقف لشم الأزهار أو لتناول كوب من القهوة.

الاستدلال هو الأسلوب الذي يساعدك في البحث عن الحل. وتكون نتائجه محض الصدفة لأن الاستدلال يخبرك فقط كيف سيبدو، وليس ما ستجده، هو لا يخبرك كيف تذهب مباشرة من النقطة (أ) إلى النقطة (ب)، حتى أنه لا يعرف أين تقع النقطة (أ) والنقطة (ب). في الواقع الاستدلال هو خوارزمية في حلة متنوعة، إنه أقل قدرة على التنبؤ، كما إنه ممتع أكثر ويُنجز في أقل من 30 يوم ويضمن استعادة التكاليف.

هنا توجد خوارزمية القيادة لمنزل شخص ما، خذ الطريق السريع 167 جنوب بويالوب (Puyallup). خذ مخرج مول التلة الجنوبي (South Hill Mall) وقد السيارة 4.5 ميل فوق التلة. استدر يمينا إلى الضوء لمخزن البقالة. ثم خذ أول يسار. استدر إلى الطريق إلى المنزل البني الكبير على اليسار، إلى 714 شمال الأرز (North Cedar).

هنا يوجد استدلال للقيادة إلى منزل شخص ما[1]: انظر إلى الرسالة الأخيرة التي أرسلناها لك. قُد السيارة إلى البلدة التي في عنوان المرسل. عندما تصل للبلدة، أسأل شخص ما أين يقع منزلنا. الجميع يعرفنا- سيكون هذا الشخص سعيد لمساعدتك. إن لم تجد أحد، كلمنا بالهاتف من هاتف عمومي، ونحن سنأتي لأخذك.

إنّ الفرق بين الخوارزمية والاستدلال غير ملحوظ. وشروط الاثنين متداخلة إلى حد ما. من أجل أهداف هذا الكتاب، الاختلاف الرئيسي بين الاثنين هو مستوى عدم المباشرة في الحل. تعطيك الخوارزمية التعليمات بشكل محدد. بينما يخبرك الاستدلال كيف تكتشف التعليمات بنفسك، أو على الأقل أين تبحث عنهم.

إنّ امتلاك الاتجاهات التي تخبرك بالضبط كيف تحل مشكلتك البرمجية سوف يجعل بالتأكيد البرمجة أسهل والنتائج أكثر تنبؤاً. لكن علم البرمجة ليس ذلك العلم المتقدم الآن وربما لن يصبح كذلك أبداً. الجزء الأكثر تحدياً في البرمجة هو تصور المشكلة. وأخطاء كثيرة في البرمجة تكون عبارة عن أخطاء مفاهيمية conceptual errors. لأن كل برنامج فريد من الناحية المفاهيمية، فمن الصعب أو المستحيل إنشاء مجموعة عامة من الوجهات التي تقود للحل في كل حالة. وبالتالي فإن معرفة كيفية التعامل مع المشكلات في العموم هو قيّم على الأقل كمعرفة الحلول محددة لمشاكل محددة.

كيف تستخدم الاستعارات البرمجية؟ استخدمهم لإعطائك رؤية للمشاكل البرمجية والعمليات. استخدمهم لمساعدتك في التفكير في أنشطتك البرمجية ومساعدتك لتخيل طرق أفضل للقيام بالأشياء. لن تكون قادراً على البحث عن سطر من البرنامج وتقول إنه انتهك واحد من الاستعارات المبينة في هذا الفصل. مع مرور الوقت، على الرغم من ذلك الشخص الذي يستخدم الاستعارات لإلقاء الضوء على عملية تطوير البرمجيات سيعتبر كشخص لديه فهم أفضل للبرمجة وإنتاج أفضل للبرنامج بشكل أسرع من الناس الذين لا يستخدموها.

3 .2 استعارات البرمجية الشائعة

لقد نمت وفرة محيرة من الاستعارات حول تطوير البرمجيات. يقول ديفد غريز(David Gries) إن كتابة البرنامج هو علم (1981).يقول دونالد كنوث(Donald Knuth) إنه فن (1998). يقول واتس هومفري(Watts Humphrey) إنه معالجة (1989). يقول بلوكر وكنت بيك (P. J. Plauger and Kent Beck) إنه يشبه قيادة السيارة، على الرغم من أنها استخلاص استنتاجات متعاكسة تقريباً (بلوكر 1993،بيك 2000). يقول أليستير كوكبرن(Alistair Cockburn) إنها لعبه (2002). يقول ايرك ريموند(Eric

[1] **إشارة مرجعية**: للحصول على التفاصيل في كيفية استخدام الاستدلال في تصميم البرمجيات، شاهد "التصميم هو معالجة استدلالية " في القسم 5.1

Raymond) إنها تشبه البازار(2000). يقول آندي هانت(Andy Hun) وديف توماس(Dave Thomas) إنها تشبه البستنه. ويقول بول هيكل (Paul Heckel) أنها تشبه تصوير بياض الثلج والأقزام السبعة (1994). يقول فريد بروكس(Fred Brooks) أنها تشبه الزراعة، أو صيد الذئاب الضارية، أو الغرق مع الديناصورات في حفرة القطران (1995). ما هي أفضل الاستعارات؟

فن كتابة البرمجيات: كتابة الشفرة

تنشأ الاستعارة الأكثر بدائية لتطوير البرمجيات من التعبير "كتابة البرنامج". توحي استعارة الكتابة أن تطوير البرنامج يشبه كتابة الرسالة العادية- حيث تجلس ومعك قلم أزرق، وحبر، وورقة، وتكتبها من البداية للنهاية. هي لا تتطلب أي تخطيط رسمي، وتجد ما تريد أن تقول كما يحلو لك.

تنبثق العديد من الأفكار من استعارة الكتابة. يقول جون بينتلي (Jon Bentley) يجب أن تكون قادراً على الجلوس بجانب النار مع كأس من البراندي، سيجار جيد، وكلب الصيد المفضل لديك لتستمتع بـ "برنامج تثقيفي" أي تهيئة ظروفك الخاصة للقيام بالعمل. سمى بريان كيرنيغان(Brian Kernighan) وبلوكر(P. J. Plauger) كتابهم نمط البرمجة بـ عناصر نمط البرمجة (The Elements of Programming Style) (1978). بعد كتاب أسلوب الكتابة المسمى عناصر النمط (The Elements of Style) (سترانك ووايت 2000) (Strunk and White 2000). غالبا ما يتكلم المبرمج عن "إمكانية قراءة البرنامج".

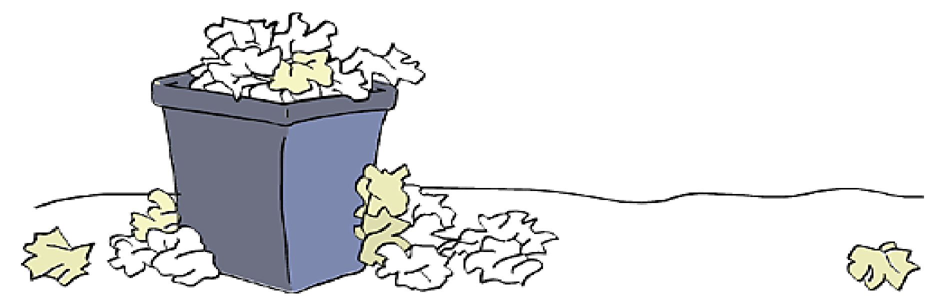
بيانات مثبتة

من أجل العمل الفردي أو من أجل المشاريع صغيرة النطاق، فإن استعارة كتابة الرسالة تعمل بشكل كافٍ، لكن من أجل أهداف أخرى فإنها لن تكون كذلك – حيث أنها لن تصف تطوير البرنامج بشكل كامل أو كافٍ. غالباً ما تكون الكتابة نشاط فردي، في حين أن مشروع البرمجيات سيشمل على الأرجح العديد من النّاس بالعديد من الإمكانيات المختلفة. عندما تنهي كتابة الرسالة، ستضعها في مغلف وترسلها. ولن تستطيع تغييرها بعد ذلك، وتعتبر أن الغاية منها قد تمت. ليس من الصعب تغيير البرنامج ولكن من الصعب اكتماله تماما. حيث أن 90 بالمئة من جهد التطوير لنظام البرمجة النموذجي يأتي بعد الإصدار الأولي. ويكون قياسيا بنسبة الثلثين (بيكوسكي 1997 (Pigosk)). في الكتابة، توضع قيمة عالية للإبداع. في بناء البرمجيات، محاولة إنشاء عمل إبداعي حقيقي غالباً ما يكون أقل فعالية من التركيز على إعادة الاستخدام لأفكار التصميم والشفرة وحالات الاختبار من المشاريع السابقة. باختصار، تتطلب استعارة الكتابة السليمة عملية تطوير برمجيات بسيطة جداً وثابتة.

لسوء الحظ، لقد استمرت استعارة كتابة الرسائل كواحدة من أكثر الكتب شعبية على الكوكب، كتاب فريد بروكس (Fred Brooks) شهر الرجل الأسطوري (The Mythical Man-Month) (بروكس 1995). يقول

بروكس " الخطة لإبعاد شخص ما، ستقوم بفعلها بأي حال."[1] هذا يستحضر صورة مجموعة من المسودات غير المكتملة المرمية إلى سلة المهملات، كما هو مبين في الشكل 2-1

الشكل 2-1 تقترح استعارة كتابة الرسالة العمليات البرمجية تتطلب اختبارات وكشف أخطاء موسّع أكثر من اعتمادها على التصميم والتخطيط الدقيق.



الشكل 2-1 تقترح استعارة كتابة الرسالة العمليات البرمجية تتطلب اختبارات وكشف أخطاء موسّع أكثر من اعتمادها على التصميم والتخطيط الدقيق.

يمكن أن يكون التخطيط عمليا لاستبعاد شخص ما. لكن توسيع مجال الاستعارة لتشمل "كتابة" برنامج للتخطيط لاستبعاد شخص ما هي نصيحة غير جيدة لتطوير البرمجيات، حيث يكلف النظام الرئيسي كما يكلف بناء مكتب ب 10 طوابق أو بناء سفينة.

إنه من السهل في لعبة الأحصنة الدوارة الإمساك بالحلقة النحاسية إن استطعت تحمّل الجلوس على حصانك الخشبي المفضل لأجل عدد غير محدود من الدورات. الحيلة هي أن تحصل عليها في المرة الأولى أو أن تأخذ فرص متعددة عندما تكون ممكنة.

الاستعارات الأخرى تسلط الضوء بشكل أفضل على طرق تحقيق مثل هذه الأهداف.

زراعة البرمجيات: تطوير النظام

بالمقارنة مع استعارة الكتابة الجامدة، يخبرك بعض مطوري البرمجيات أنه يجب تصور إنشاء البرنامج كشيء ما يشبه زراعة البذور وجني المحاصيل. أنت تصمم قطعة، وتشفر قطعة، وتختبر قطعة، وتضيفها إلى النظام بشكل تراكمي في كل مرة. باتخاذ خطوات صغيرة، بالتالي فإنك تقلل المتاعب التي يمكن أن تتورط فيها في أي وقت من الأوقات.

[1] " الخطة لإبعاد شخص ما، ستقوم بفعلها بأي حال." –فريد بروكس

إذا خططت لاستبعاد شخص ما، ستبعد أنت أيضا- كرايك زيروني



في بعض الأحيان توصف تقنية جيدة باستعارة سيئة. في مثل هذه الحالات، حاول أن تحافظ على التقنية وتخرج بأفضل استعارة. في هذه الحالة، تكون التقنية الإضافية قيمة، لكن استعارة الزراعة سيئة.[1]

الفكرة من إنجاز القليل في كل مرة ربما تبدو تحمل بعض التشابه لطريقة جني المحاصيل، لكن التشابه الزراعي ضعيف وغير مفيد، ومن السهل استبداله بالاستعارات الأفضل الموصوفة في المقطع التالي. إنه من الصعب تمديد الاستعارات الزراعية لأبعد من الفكرة البسيطة لفعل الأشياء بشكل قليل في كل مرة. إذا اشتريت حصة في الاستعارة الزراعية، مبين في الشكل 2-2، من المحتمل أنك ستجد نفسك تتحدث عن تخصيب خطة النظام وترقيق التصميم التفصيلي وزيادة غلة الشفرة من خلال إدارة فعالة للأرض، وحصد الشفرة نفسها. سوف تتحدث عن التناوب في محصول سي ++ بدلا من الشعير، لتجعل الأرض ترتاح لسنة لزيادة التزود بالنتروجين في القرص الصلب.

الضعف في استعارة زراعة البرمجيات هي اقتراحها أنك لا تملك أي تحكم مباشر أكثر من كيف تتطور البرمجيات. تزرع بذور الشفرة في الربيع. ووفقاً لتقويم الفلاح فإنك على موعد مع المحصول الجيد، حيث أنك ستملك محصولاً وافراً من الشفرة في الخريف.



الشكل 2-2 إنه من الصعب تمديد استعارة الزراعة لتطوير البرمجيات بشكل مناسب

برمجية زراعة المحار: تزايد النظام التراكمي

في بعض الأحيان يتكلم الناس عن البرمجيات المتنامية عندما يقصدون حقيقةً تراكم البرمجيات. الاستعارتين مرتبطتين بشكل وثيق، ولكن تراكم البرمجيات هو التصور الأكثر وضوحاً. "التراكم"، في حال لم يكن لديك قاموس مفيد، فهو يعني أي نمو أو تزايد في الحجم عن طريق تضمين أو إضافة خارجية تدريجية. يصف التراكم الطريقة لجعل المحار لؤلؤ، بإضافة كميات صغيرة من كربونات الكالسيوم تدريجياً. في الجيولوجيا، "

[1] **مزيد من القراءة** من أجل توضيح الاستعارات الزراعية المختلفة، الاستعارة التي ستطبق لصيانة البرمجيات، شاهد الفصل "على مصادر حدس المصمم" في إعادة التفكير في أنظمة التصميم والتحليل (وين بيرغ Wein berg 1988)

التراكم" يعني إضافة بطيئة إلى الأرض عن طريق إيداع الرواسب المنقولة في الماء. من الناحية القانونية، "التراكم" يعني تزايد اليابسة على طول شواطئ الرقعة المائية عن طريق الرواسب المنتقلة عن طريق المياه.

هذا لا يعني أنه يجب عليك تعلم كيف تصنع الشفرة خارج الرواسب التي تنتقل في المياه، إنه يعني أنه يجب عليك تعلم كيف تضيف لنظامك البرمجي كميات قليلة في كل مرة. يوجد كلمات أخرى مرتبطة بشكل وثيق بالتراكم " تدريجي"، "متكرر"، "متكيف" و"متطور". التصميمات الإضافية، والبناء، والاختبار هم بعض أقوى المفاهيم المتاحة لتطوير البرمجيات.[1]

في التطوير المتزايد، تقوم أولا بعمل النسخة المحتملة الأبسط من النظام التي سيتم تشغيلها. حيث أنها لا تفرض عليك مدخلات واقعية، ولا تؤدي إلى مناورات واقعية على البيانات، كما لا تنتج خرج واقعي ــ يجب أن تكون فقط هيكل قوي كفاية ليحافظ على النظام الحقيقي كما وضع. يمكن أن يستدعى ذلك صفوف (classes) وهمية لأجل كل من الوظائف الأساسية التي قمت بتحديدها. تشبه هذه البداية الأساسية بداية لؤلؤ المحار مع حبات صغيرة من الرمل.

بعد أن تشكل الهيكلية العامة (الهيكل العظمي)، تضع شيئا فشيئا العضلات والجلد، تقوم بتغيير كل الصفوف الوهمية إلى صفوف حقيقية. بدلاً من ان يتظاهر برنامجك بقبول المدخلات، تسقطه عملياً على الشفرة الذي يقبل دخلاً حقيقياً. بدلاً من أن يدعي برنامجك إنتاج خرج، تسقطه على الشفرة الذي يعطي خرجاً حقيقياً. وتضيف كمية قليلة من التعليمات البرمجية في كل مرة حتى تمتلك نظام عمل كامل.

إن الأدلة المتناقلة المؤيدة لهذا النهج رائعة. فريد بروكس، الذي نصح ببناء واحد للمضي قدماً، قال إنه لا شيء في العشر سنوات بعدما كتب كتابه التاريخي " شهر الرجل الأسطوري" غيّر بشكل جذري تمارينه الخاصة أو فعاليتها كتطور متزايد (1995). أنجز توم غلب (Tom Gilb) نفس النقطة في كتابه المتقدم "مبادئ إدارة هندسة البرمجيات" (Principles of Software Engineering Management)(1988).الذي أنتج تنفيذ متطور ووضع الأساس للكثير من نهج البرمجة الرشيقة اليوم. تعتمد العديد من المنهجيات الحالية على هذه الفكرة. (Beck 2000، Cockburn 2002، Highsmith 2002، Reifer 2002، Martin 2003، Larman 2004).

كاستعارة، قوة الاستعارة المتزايدة في أنها لا تقوم على المبالغة كما أنها أمتن من استعارة الزراعة لتشبيه غير ملائم. أن صورة تشكيل المحار للؤلؤ هي طريقة جيدة لتصور التطوير المتزايد أو المتراكم.

[1] **إشارة مرجعية** للحصول على تفاصيل عن كيفية تطبيق الاستراتيجيات المتزايدة لتكامل النظام، شاهد الفصل 29.2," تردد التكامل " ــ مرحلي أو تزايدي

تشييد البرمجيات: بناء البرنامج

نقطة مفتاحية

إن مشهد "بناء" البرنامج أكثر نفعاً من "كتابة" أو "نمو" البرنامج. فإنه متوافق مع فكرة التراكم البرمجي ويعطي إرشادات تفصيلية أكثر. يتطلب بناء البرنامج مراحل مختلفة من التخطيط، والإعداد، والتنفيذ الذي يختلف بالنوع والدرجة معتمداً على ما يجري بناءه. عندما تجد الاستعارة المناسبة، ستجد العديد من أوجه التشابه الأخرى.

بناء برج من أربعة أقدام يتطلب يداً ثابتة، وسطحا مستوياً، و10 علب بيرة غير تالفة. بناء برج يزيد بـ 100 مرة من هذا الحجم لا يتطلب فقط العديد من علب البيرة أكثر بـ 100مرة. بل إنه يتطلب نوعا مختلفا تماما من التخطيط والبناء.

إن كنت تبني بناءً بسيطاً ـ منزل كلب ـ تقول، يمكنني أن أقود السيارة إلى مخزن الخشب وأشتري بعض الخشب والمسامير. في آخر المساء، سيصبح لدي بيتاً جديداً من أجل كلبي "فيدو". إن نسيت تأمين الباب، المبين في الشكل 2-3، أو وقعت ببعض الأخطاء الأخرى، ليست مشكلة كبيرة فأنت تستطيع إصلاحها أو حتى تبدأ من البداية. كل ما أضعته هو جزء من فترة المساء.

هذا المنهج الحر مناسب للمشاريع البرمجية الصغيرة أيضاً. إن استخدمت التصميم الخاطئ من أجل 1000 سطر من التعليمات البرمجية، فإنه بالإمكان إعادة التصنيع أو البدء من جديد بشكل كامل بدون خسارة الكثير.

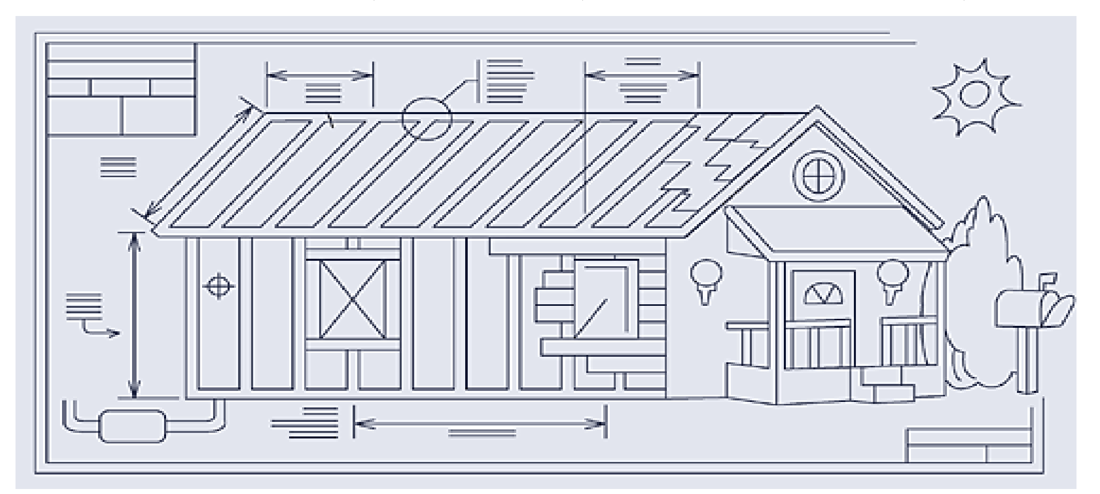


الشكل 2-3 العقوبة على الخطأ في بناء بسيط هي فقط وقت قليل وربما بعض الحرج

أما إذا كنت تبني منزلاً، فإن عملية البناء ستكون أكثر تعقيداً، وكذلك ستكون عواقب سوء التصميم. أولاً يجب أن تقرر ما هو نوع المنزل الذي تريد بناءه ـ يقابله تحديد المشكلة في تطوير البرمجيات. ثم يجب عليك أنت والمهندس المعماري التوصل إلى تصميم عام واعتماده. هذا مشابه للتصميم الهندسي البرمجي. ترسم مخططات مفصلة وتستأجر مقاول. وهذا مشابه لتصميم البرمجيات المفصلة. تُعد موقع البناء، وتضع الأساس، وإطار المنزل، وتضع جدران خارجية وسقف عليها، وسمكرتها وتمديد الأسلاك، هذا مشابه لتشييد البرمجيات. عندما

ينجز معظم المنزل، فإن منسقي الحدائق، والدهانين، ومصممي الديكورات يقومون بالأفضل لممتلكاتك وللمنزل الذي بنيته. هذا مشابه للاستخدام الأمثل للبرمجيات. خلال العملية، يأتي مراقبين متنوعين لفحص الموقع، والأساس، والهيكل، والأسلاك، وأشياء أخرى. هذا مشابه لمعاينة ومراقبة البرامج.

زيادة التعقيد والحجم تعني ضمناً عواقب أكبر في كلا النشاطين. في بناء منزل، تكون المواد مكلفة إلى حد ما، لكن النفقات الرئيسية هي اليد العاملة. إن انتزاع جدار وتحريكه 6 إنشات مكلف ليس لأنك أهدرت العديد من المسامير، لكن لأنك يجب أن تدفع للناس الوقت الإضافي الذي يستغرقه تحريك الحائط. ينبغي جعل التصميم جيداً قدر الإمكان، كما يوحي الشكل 2-4، وبذلك لا تهدر الوقت في إصلاح الأخطاء التي كان من الممكن تجنبها. في بناء المنتج البرمجي، تكون المواد أقل تكلفة، لكن اليد العاملة مكلفة أكثر. تغيير شكل التقرير يكون مكلف كتحريك الجدار في المنزل، لأن التكلفة الرئيسية في كلتا الحالتين هي وقت الناس.



الشكل 2-4 أبنية أكثر تعقيدا تتطلب تخطيط دقيق أكثر

ما هي أوجه التشابه الأخرى التي يتقاسمها النشاطان؟ في بناء المنزل، لن تحاول بناء الأشياء التي تستطيع شراءها والمبنية مسبقاً. ستشتري الغسالة والمجففة، وغسالة الصحون، والثلاجة، والمجمدة. ما لم تكن أنت ميكانيكياً، لن تأخذ بعين الاعتبار بنائهم بنفسك. سوف تشتري أيضاً الخزن الجاهزة، والطاولات، والنوافذ، وتجهيزات الحمام. إن كنت تبني نظاماً برمجياً، ستفعل نفس الشيء. ستستخدم على نطاق واسع ميزات لغة عالية المستوى بدلاً من كتابة الشفرة الخاصة بمستوى نظام التشغيل الخاص بك. وقد تستخدم أيضاً مكتبات تم إنشاءها مسبقاً من الصفوف المُضمّنة (container classes)، والوظائف العلمية، وأصناف واجهة المستخدم، وأصناف معالجة قواعد البيانات. في العموم ليس منطقياً كتابة الشفرة للأشياء التي يمكنك شراءها جاهزة.

إذا كنت تبني بيتاً فخماً مع أساس من النخب الأول، قد يكون لديك خزائنك المصنوعة حسب الطلب. قد يكون لديك غسالة صحون، وثلاجة، ومجمدة، قد صُنعت لتبدو وكأنها تشبه بقية خزائنك. قد يكون لديك نوافذ

مصنوعة حسب الطلب في حجوم وأشكال غير عادية. لهذه المواصفات أوجه شبه في تطوير البرمجيات. أن تبني منتج برمجي من الدرجة الأولى، يمكنك بناء وظائفك العلمية الخاصة من أجل دقة وسرعة أفضل. يمكنك بناء الصفوف المُضمنة الخاصة بك وصفوف واجهة المستخدم، وصفوف قاعدة البيانات لتعطي نظامك المرونة، وانسجام تام في النظر والاحساس.

يمكن لكل من تشييد البناء وتشييد البرمجيات الاستفادة من المستويات المناسبة من التخطيط. إن بنيت برنامج بأوامر خاطئة، فسوف يكون من الصعب كتابة التعليمات البرمجية، ومن الصعب الاختبار، ومن الصعب التصحيح. ويمكن أن يستغرق وقتا أطول ليكتمل، أو سينهار المشروع لأن عمل كل شخص معقد جداً وبالتالي سيكون مربكاً جداً عندما يجتمع كل ذلك.

لا يعني التخطيط الدقيق بالضرورة التخطيط الشامل أو التخطيط الزائد. يمكنك التخطيط خارج الدعائم الهيكلية والتقرير فيما بعد فيما إذا كنت ستضع الأرضيات الخشبية الصلبة أو السجاد، أي لون ستختار لدهن الجدار، ما هي مواد السطح المستخدمة، وهكذا، يحسّن المشروع الذي تم التخطيط له جيداً قدرتك على تغير تفكيرك حول التفاصيل فيما بعد. كلما زادت خبرتك بنوع البرمجيات التي تبنيها، زادت التفاصيل التي يمكن اتخاذها بشكل مفروغ منه. يجب فقط أن تكون متأكداً أنك تخطط بشكل كاف لكيلا يشكل نقص التخطيط مشاكل كبيرة فيما بعد.

يساعد تشبيه البناء أيضاً في توضيح لماذا تستفيد المشاريع البرمجية المختلفة من مناهج تطوير مختلفة. في البناء، ستستخدم مستويات مختلفة من التخطيط، والتصميم، وضمان الجودة إن كنت تبني مستودع أو معمل عن تلك التي ستكون لو كنت تبني مركز طبي أو مفاعل نووي. سوف تتعاطى مناهج مختلفة لبناء مدرسة، أو ناطحة سحاب، أو منزل من ثلاث غرف. بالمثل، من المحتمل بشكل عام في البرمجيات استخدام مناهج تطوير بسيطة وثابتة، لكن ستحتاج في بعض الأحيان مناهج معقدة وجامدة لتحقيق أهداف آمنة، وأهداف أخرى.

يجلب إجراء تغيرات في البرنامج تشبيها موازٍ مع تشييد البناء. يكلف تحريك الجدار 6 إنشات أكثر إذا كان على الجدار حمل فيما لو كان مجرد تقسيم بين الغرف. بشكل مشابه، يكلف إجراء تغيرات بنيوية في البرنامج أكثر من إضافة أو حذف السمات الجانبية.

في النهاية، تُلقي استعارة البناء الضوء على المشاريع البرمجية الكبيرة جداً. لأن عاقبة الفشل في البناء الكبير جداً شديدة. البناء يجب أن يكون فائق الهندسة. يصنع ويفحص البناؤون خططهم بعناية. يبنون بهامش أمان، من الأفضل أن تدفع عشر بالمئة أكثر من أجل مواد أقوى من أن تملك ناطحة سحاب قد تسقط. يُعطى قدر كبير

من الاهتمام للتوقيت. عندما بني بناء "ايمباير ستيت"(Empire State)، كان لكل شاحنة هامش تسليم 15 دقيقة لتنجز التسليم بها. إن لم تكن الشاحنة في المكان خلال التوقيت الصحيح، تأخر المشروع بأكمله.

بالمثل، يتطلب لأجل المشاريع البرمجية الكبيرة جداً تخطيطاً لترتيبات بمستوى أعلى، مقارنة بالمشاريع التي هي مجرد كبيرة. إن تقارير "كيبر جونز" (Capers Jones) أن نظام برمجي بمليون سطر من التعليمات البرمجية يتطلب بمعدل 69 نوع من الوثائق (1998). مواصفات المتطلبات لمثل هذا النظام ستكون عادة حوالي 4000-5000 صفحة، ويمكن أن تكون وثائق التصميم ببساطة حوالي مرتين أو ثلاث مرات أكبر من المتطلبات. فمن غير المرجّح أن يكون الفرد قادراً على فهم التصميم الكامل لمشروع من هذا الحجم أو حتى قراءته. لذلك فإن درجة كبيرة من الإعداد ستكون مناسبة.

لقد قمنا ببناء مشاريع برمجية قابلة للمقارنة في الحجم الاقتصادي لبناء "ايمباير ستيت"، وتطلبت ضوابط إدارية وفنية مشابهه.

يمكن أن تتمدد استعارة تشييد البناء في اتجاهات أخرى متنوعة[1]، وهذا هو السبب في كونها استعارة فعالة جداً. العديد من المصطلحات المشتركة في تطوير البرمجيات مستمدة من استعارة البناء: هندسة البرمجيات، والدعامات، والتشييد، وطبقات الأساس، وتجزئة البرنامج لأجزاء. ومن المحتمل أن تسمع بالمزيد.

تطبيق تقنيات البرمجيات: صندوق الأدوات الذهنية



قد أمضى الناس الفاعلين في مجال تطوير البرمجيات العالية النوعية سنوات في تجميع عشرات التقنيات، والحيل، والتمائم السحرية (الشفرات الذكية). هذه التقنيات ليست قواعد بل هي أدوات تحليلية. الحرفي الجيد يعرف تماماً الأداة المناسبة للعمل وكيفية استخدامها بشكل صحيح. كما يفعل المبرمجون ذلك أيضاً. فكلما تعلمت عن البرمجة، كلما ملأت صندوق أدواتك الذهنية بالأدوات التحليلية والمعرفة بتوقيت وكيفية استخدامها بالشكل الصحيح.

في البرمجة، ينصحك الخبراء الاستشاريون أحيانا بأن تقبل بطرق برمجية محددة وتستبعد طرق أخرى[2]. وهذا غير مناسب لأنه إذا كنت تعمل وفق منهجية واحدة 100 بالمئة، فإنك سوف ترى كل العالم حسب تلك المنهجية.

[1] **مزيد من القراءة** من أجل بعض التعليقات الجيدة عن تمدد استعارة البناء، شاهد What Supports the Roof ? (ستار2003)

[2] **إشارة مرجعية** لتفاصيل عن اختيار وجمع الأساليب في التصميم، شاهد الفصل 5.3، "تصميم اللبنات الأساسية: الاستدلال"

وفي بعض الحالات، ستخسر الفرص لاستخدام أفكار أخرى مناسبة أكثر لمشكلتك الحالية. إن استعارة الأدوات تساعد في الحفاظ على كل الأساليب، والتقنيات، والنصائح الجاهزة للاستخدام عند الحاجة إليها.

دمج الاستعارات

نقطة مفتاحية

لأن الاستعارات كاشفه أكثر منها خوارزمية، فهي لا تستبعد بعضها البعض. يمكنك استخدام كل من استعارات التشييد والتراكم. يمكنك استخدام استعارة الكتابة إن رغبت في ذلك، ويمكنك الدمج بين الكتابة والقيادة، أو صيد الذئاب الضارية، أو الغرق مع الديناصورات في حفرة من القطران. استخدم أي استعارة أو مجموعة من الاستعارات تحفز تفكيرك الخاص أو تجعل التواصل مع الآخرين في فريقك بشكل جيد.

إن استخدام الاستعارات هو عمل غير واضح. يجب عليك توسعتهم للاستفادة من الرؤى الاستدلالية التي تمنحها. لكن إن قمت بتوسيعهم بشكل كبير أو في الاتجاه الخاطئ، فإنهم سيضللونك. تماماً كما يمكنك إساءة استخدام أداة قوية، يمكنك إساءة استخدام الاستعارات، لكن قوتهم تجعلهم الجزء القيّم من أدواتك الذهنية.

مصادر إضافية

من بين الكتب العامة عن الاستعارات، الأنماط، والنماذج، كتاب المحك " the touchstone book" لـ توماس كون (Thomas Kuhn).

- كون، توماس س. بنية الثورات العلمية:

Kuhn، The Structure of Scientific Revolutions. Thomas S، 3d ed. Chicago، IL: The University of Chicago Press، 1996.

كتاب كون عن كيفية ظهور الثورات العلمية، تطورها، وخضوعها لنظريات أخرى في الحلقة الداروينية، وضع فلسفة العلم على السمع عندما انتشرت أولا في عام 1962 وهي واضحة وقصيرة، ومحملة بأمثلة ممتعة من رقي وهبوط الاستعارات والأنماط والنماذج في العلم.

- فلويد، روبرت "نماذج البرمجة"

Floyd، Robert W. “The Paradigms of Programming.” 1978 Turing Award Lecture. Communications of the ACM، August 1979، pp. 455–60.

وفيه مناقشة رائعة للأنماط في تطوير البرمجة، وأضاف فلويد أفكار كون للموضوع.

نقاط مفتاحية

- الاستعارات هي استدلال، ليست خوارزمية، على هذا النحو، تميل لأن تكون مهملة قليلاً.
- تساعدك الاستعارات على فهم عملية تطوير البرمجيات عن طريق ربطها بنشاطات أخرى أنت تعرف عنها بالفعل.
- بعض الاستعارات أفضل من الاستعارات الأخرى.
- إن معالجة بناء البرمجيات بشكل مشابه لتشييد البناء يوحي بأنه بحاجة لإعداد دقيق وإلقاء الضوء على الفرق بين المشاريع الكبيرة والصغيرة.
- التفكير في ممارسات تطوير البرمجيات كأدوات في صندوق أدوات ذهنية يقترح كذلك أن كل مبرمج يملك أدوات متعددة وأنه لا يوجد أداة مفردة صحيحة لكل مهنة. اختيار الأداة الصحيحة لكل مشكلة هو المفتاح الأول لتصبح مبرمج فعّال.
- الاستعارات لا تلغي بعضها البعض. استخدم مزيجاً من الاستعارات التي ستقوم بتقديم الأفضل.

الفصل 3

قس مرتين، اقطع مرة: المتطلبات الأولية التحضيرية

المحتويات

مواضيع ذات صلة

- قرارات البناء الرئيسية: الفصل 4
- تأثير حجم المشروع على البناء والمتطلبات الأولية: الفصل 27
- العلاقة بين أهداف الجودة وأنشطة البناء: الفصل 20
- إدارة البناء: الفصل 28
- التصميم: الفصل 5

قبل البدء ببناء منزل، يراجع الباني المخططات، ويتحقق من الحصول على كافة التراخيص اللازمة، ويدرس أساسات المنزل. حيث يُحضّر الباني لإقامة ناطحة سحاب بطريقة وللتطوير السكني بطريقة أخرى، ولبناء منزل للكلب بطريقةٍ ثالثة. مهما كان المشروع، فالتخطيط يُضبط لتلبية متطلبات المشروع الخاصة ويحضّر بأمانة قبل بدء البناء.

يصف هذا الفصل العمل المطلوب لإعداد بناء البرمجيات. كما هو الحال مع تشييد البناء، فإن الكثير من حالات نجاح أو فشل المشروع تتحدّد قبل أن يبدأ البناء. إذا لم يوضع الأساس بشكل صحيح أو كان التخطيط غير ملائم، فأفضل ما يمكنك القيام به أثناء البناء هو الحفاظ على الضرر بالحد الأدنى.

إن قول النجار: " قس مرتين، اقطع مرة" ينطبق بشكل كبير على الجزء المتعلق بالبناء في تطوير البرمجيات، والذي يمثل أكثر من 65% من تكلفة المشروع الكليّة. أسوأ المشاريع البرمجية تلك التي قد تنتهي بالقيام بالبناء مرتين أو ثلاث مرات وربما أكثر. إنّ فكرة إعادة أكثر الأجزاء تكلفة في المشروع مرتين هي فكرة سيئة في البرمجيات كما هو الحال في أي مجال آخر.

على الرغم من أن هذا الفصل يضع الأساس لبناء ناجح للبرمجيات، إلّا أنه لا يناقش البناء بشكل مباشر. إذا كنت تشعر بنفسك ضليعاً وعلى دراية جيدة بمراحل حياة هندسة البرمجيات، ستجد بداية البناء الأساسية "الزبدة" في الفصل 5 "التصميم في البناء". استعرض القسم 2.3 إذا لم تحب فكرة المتطلبات الأولية للبناء، لتعرف كيفية تطبيق الشروط المسبقة على حالتك" حدد نوع البرمجيات التي تعمل عليها"، ومن ثم خذ نظرة على البيانات في القسم 1.3، الذي يصف تكلفة عدم إنجاز الشروط المسبقة.

1.3 أهمية المتطلبات الأولية

القاسم المشترك للمبرمجين الذين يبنون برمجيات عالية الجودة هو استخدامهم لأنشطة عالية الجودة. تركز مثل هذه الأنشطة على الجودة في كل مراحل المشروع، في بداية المشروع، ومنتصفه، وفي نهايته.[1]

إذا كنت تركز على الجودة في نهاية المشروع، فهذا يعني أنك تركز على اختبار النظام. الاختبار هو ما يفكر به الكثير من النّاس عندما يفكرون بالتأكد من ضمان جودة البرمجيات. على أية حال، الاختبار هو جزء واحد فقط من استراتيجية ضمان الجودة الكامل، وهو ليس الجزء الأكثر تأثيراً. حيث لا يمكن للاختبار اكتشاف عيب كبناء المنتج الخاطئ أو بناء المنتج الصحيح بطريقة خاطئة. مثل هذه العيوب يجب أن تكتشف بوقت سابق قبل الاختبار ـ وقبل أن يبدأ البناء.



وإذا كنت تركز على الجودة في منتصف المشروع، فإنك إذاً تركز على أنشطة البناء. مثل هذه الأنشطة ستكون محور معظم هذا الكتاب.

أما إذا كنت تركز على الجودة في بداية المشروع، فإنك تخطط لمتطلب، وتصميم منتج عالي الجودة. فإذا بدأت العملية بتصميم من طراز أزتيك (Pontiac Aztek)، يمكنك اختبارها كل ما أردت، لكن لن يتحول إلى طراز

[1] **إشارة مرجعية** إن الانتباه للجودة هو الطريقة الأفضل لتحسين الإنتاجية. للتفاصيل، انظر القسم 5.20 "المبدأ العام لجودة البرمجيات"

رولز- رويس(Rolls-Royce). يمكن أن تبني أفضل سيارة أزتيك ممكنة ولكن إذا أردت روزلز رويس فيجب أن تخطط لذلك منذ البداية. في تطوير البرمجيات تقوم بمثل هذا التخطيط عندما تعرّف المشكلة وعندما تحدد الحل وعندما تصمم الحل.

طالما أن البناء في منتصف المشروع البرمجي، فإنك مع الوقت ستصل الى البناء، والأجزاء الأولى السابقة من المشروع تكون قد هيئت الأرضية للنجاح أو الفشل. على أية حال، يجب على الأقل أن تكون قادراً خلال البناء على تحديد جودة حلك وأن تتراجع عندما ترى غيوم الفشل السوداء تلوح في الأفق. تصف بقية هذا الفصل بالتفصيل أهمية الإعداد السليم وتخبرك كيف تحدد فيما إذا كنت جاهزاً بالفعل لبدء البناء.

هل تنطبق الشروط المسبقة على مشاريع البرمجة الحديثة؟[1]

يؤكد البعض أن الأنشطة التحضيرية كالهيكلة، والتصميم، وتخطيط المشروع غير مفيدة في مشاريع البرمجة الحديثة. إن مثل هذه الادعاءات حقيقةً غير مدعومة جيداً بالبحوث، لا في الماضي ولا الحاضر ولا في البيانات الحالية. (شاهد بقية هذا الفصل للحصول على التفاصيل).

يعتبر المعارضون للشروط المسبقة عادةً أن أمثلة الشروط المسبقة التي أُنجزت سيئة ويشيرون أيضاً إلى أن هذا العمل غير فعّال.

الأنشطة التحضيرية يمكن أن تنجز بشكل جيد، وبكافة الأحوال، تشير بيانات الصناعة من سبعينيات القرن العشرين حتى يومنا الحاضر إلى أن تلك المشاريع كانت لتنفّذ بشكل أفضل فيما لو أنجزت أنشطة الشروط المسبقة المناسبة قبل بدء البناء بشكل جدي.

إن الهدف العام للإعداد هو الحد من المخاطر: يزيل مُخطِط المشروع الجيد المخاطر الرئيسية في أقرب وقت ممكن قبل بدء المشروع، مما يمكن المشروع من المضي بأكبر قدر من السلاسة. وحتى الآن المخاطر الأكثر شيوعاً في تطوير البرمجيات هي المتطلبات السيئة والتخطيط السيء للمشروع. وبالتالي يميل الإعداد للتركيز على تحسين المتطلبات وخطط المشاريع.

الإعداد للبناء هو ليس علم دقيق، والنهج المتبع للحد من المخاطر يجب أن يقرره المشروع بنفسه. يمكن أن تختلف التفاصيل بشكل كبير بين المشاريع. للمزيد عن هذا الموضوع، شاهد القسم 2.3.

[1] المنهجية المستخدمة يجب أن تكون مبنية على خيار الأحدث والأفضل وليس على أساس الجهل. يجب أيضا أن تكون مرتبطة مع المنهجية القديمة والموثوقة. هارلن ميلز (Harlan Mills)

أسباب الإعداد غير الكامل

قد تعتقد أن جميع المبرمجين المحترفين يعرفون أهمية الإعداد والتحقق من أن الشروط المسبقة قد تم تأمينها قبل الانتقال إلى البناء. ولكن للأسف، فالوضع ليس كذلك.

السبب الشائع للإعداد غير المتكامل هو أن المطورين الذين كلفوا بالعمل على الأنشطة الأولية لا يملكون الخبرة للقيام بمهامهم المكلفين بها[1]. حيث أن المهارات المطلوبة لتخطيط مشروع، وإنشاء حالة عمل مقنعة، وتطوير متطلبات شاملة ودقيقة، وإنشاء هيكلية عالية الجودة هي ليست من البديهيات لديهم. معظم المطورين لم يتلقوا التدريب على كيفية تنفيذ هذه الأنشطة. وعندما لا يعرف المطورون كيفية القيام بالعمل الأولي، فإن التوصية "الإنجاز أعمال أولية أكثر" ستبدو كلاماً فارغاً: إن لم ينفذ العمل بشكل جيد في الموقع الأول، فإن استمرارية العمل ستكون غير مفيدة! إنّ توضيح كيفية أداء هذه الأنشطة هو خارج نطاق هذا الكتاب، لكن توفر أقسام "مصادر إضافية"[2] في نهاية هذا الفصل العديد من الخيارات للحصول على تلك الخبرة.

بعض المبرمجين يعرفون كيفية تنفيذ الأنشطة الأولية، لكنهم لا ينفذوها لأنهم لا يستطيعون مقاومة الرغبة في بدء كتابة التعليمات البرمجية في أقرب وقت متاح. إذا اخترت هذا الطريق، لدي اقتراحين. الاقتراح الأول: قراءة النصائح في القسم التالي. فقد تخبرك ببعض الأشياء التي لم تفكر بها. الاقتراح الثاني: الانتباه من المشاكل التي واجهتها سابقاً. لتتجنب الكثير من الإجهاد خلال عملك تحتاج لتعلم عدد قليل من البرامج الكبيرة فقط. دع تجربتك الشخصية تكون دليلك.

السبب النهائي في أن المبرمجين لا يعدّون الأنشطة الأولية، هو أن المدراء معروفين بعدم التعاطف مع المبرمجين الذين يقضون الوقت على إعداد المتطلبات الأولية للبناء. لقد نادى أناس مثل باري بويم (Barry Boehm)، غرادي بوش (Grady Booch)، وكارل ويغرز (Karl Wiegers) بالمتطلبات والتصميم لمدة 25 سنة، وقد تتوقع أن المدراء سيبدؤون الفهم بأن تطوير البرمجيات هو أهم من كتابة التعليمات البرمجية.

منذ عدة سنوات مضت[3]، وبينما كنت أعمل على مشروع وزارة الدفاع الذي كان يركز على تطوير المتطلبات، عندها أتى جنرال الجيش المسؤول عن المشروع للزيارة. لقد أخبرناه حينها أننا نطور المتطلبات ونتحدث بشكل

---

[1] **مزيد من القراءة** لوصف برنامج التطوير المهني الذي ينمي هذه المهارات، شاهد الفصل 16 من تطوير البرمجيات المحترفة (ماكونيل McConnell بها.

[2] cc2e.com/0316

[3] **مزيد من القراءة** لأجل العديد من التغيرات المسلية عن هذا الموضوع، أقرأ نظرية جيرالد وينبيرغ الكلاسيكية، فلسفة برمجة الحاسوب (The Psychology of Computer Programming) (Weinberg 1998)

واضح مع عملائنا، ونجمع المتطلبات، ونرسم التصميم. لكنه أصرّ على رؤية الشفرة البرمجية بأي طريقة. أخبرناه أنه لا توجد شفرة برمجية بعد، لكنه مشى حول المكان الذي يحوي 100 شخص، وعزم على إيجاد مبرمج ما. أُحبط برؤية الكثير من الناس بعيداً عن مكاتبهم أو يعملون على المتطلبات والتصميم، فأشار الرجل صاحب الصوت العالي في نهاية جولته الواسعة إلى المهندس الذي يجلس بجانبي وصرخ "ماذا يفعل؟ يجب أن يقوم بكتابة الشفرة البرمجية!" في الحقيقة، أراد من المهندس أن يعمل على الشفرة، وكان المهندس يعمل على جدوى تصميم الوثائق، لكن الجنرال أراد أن يجد الشفرة، اعتقد أنها كانت تبدو مثل الشفرة البرمجية، لذلك أخبرناه أنها الشفرة.

هذه الظاهرة تعرف باسم ويسكا (WISCA) أو متلازمة ويمب (WIMP): لماذا لم يكتب سام أي شفرة برمجية؟ أو لماذا لم تبرمج ماري؟

إذا تظاهر مدير مشروعك بالقيام بدور العميد وأمرك بالبدء بكتابة الشفرة على الفور، فإنه من السهل القول، "نعم، سيدي!" (ما المشكلة؟ يجب على الرجل القديم أن يعرف عن ماذا يتحدث). وهذه استجابة سيئة، وسيكون لديك بدائل عديدة أفضل منها. أولاً، إذا كانت علاقتك مع رئيسك وحسابك المصرفي جيدين كفاية، يمكنك الرفض القاطع بالعمل في أمر غير فعال. حظاً طيباً.

بديل آخر مشكوك به وهو التظاهر بأنك تكتب شفرة برمجية في حين أنك لا تفعل. ضع قائمة البرامج القديمة على زاوية مكتبك. ثم اذهب مباشرة وطوّر متطلباتك وهيكلتك، بموافقة رئيسك أو بدون موافقته. ستنجز المشروع بشكل أسرع وبنتائج عالية الجودة. يجد بعض الناس أن هذا النهج غير مقبول من الناحية الأخلاقية، لكن حسب وجهة نظر رئيسك، فان الجهل سيكون نعمة.

ثالثاً، يمكنك تثقيف رئيسك بالفروق الدقيقة للمشاريع الفنية. فمن الممكن أن يكون هذا النهج جيد لأنه يسهم في زيادة عدد الرؤساء المستنيرين في العالم. يعرض القسم الفرعي التالي مبرراً منطقياً لأخذ الوقت اللازم لعمل الشروط المسبقة قبل البناء.

أخيراً، يمكن أن تجد عمل آخر. على الرغم من التأرجح الاقتصادي. يكون المبرمجين الجيدين عادة لديهم نقص في العرض (BLS 2002)، والحياة قصيرة جداً للعمل في متاجر البرمجة غير المدروسة عندما يتوفر الكثير من البدائل الأفضل.

حجة مضمونة ومقنعة جداً لعمل المتطلبات الأساسية قبل البناء

افترض أنك قد وصلت إلى القمة في تحديد المشكلة، مشيت ميل مع رجل المتطلبات، تخلصت من ملابسك المتسخة في نافورة الهيكلة، وتحممت في مياه صافية من الإعداد. ستعلم عندها أنه قبل تنفيذ النظام، تحتاج لفهم ما المفترض من النظام القيام به وكيف يُفترض أن يفعل ذلك.



جزء من مهمتك كمهندس فني هو أن تثقف الناس غير الفنيين حولك عن عملية التطوير. سيساعدك هذا القسم في التعامل مع المدراء والرؤساء الذين لا يعرفون ذلك. إنه حجة كبيرة لعمل المتطلبات والهيكلة- التي تعطي الجوانب الحيوية بشكل صحيح - قبل أن تبدأ كتابة الشفرة البرمجية، والاختبار، والتصحيح.

تعلم الحجة، ثم اجلس مع رئيسك وتكلم من القلب للقلب حول عملية التطوير.

العودة إلى المنطق

أحد الأفكار الرئيسية في البرمجة الفعالة هي أهمية الإعداد. حيث يبدو منطقياً قبل البدء بالعمل على مشروع كبير أنه يجب عليك أن تخطط للمشروع. المشاريع الكبيرة تتطلب تخطيطاً أكثر في حين أن المشاريع الصغيرة تتطلب تخطيطاً أقل. من وجهة نظر الإدارة، التخطيط يعني تحديد كمية الوقت، وعدد الأشخاص، وعدد الحواسب التي سيحتاجها المشروع. ومن وجهة النظر الفنية، فالتخطيط يعني فهم ما تريد بناءه وهكذا فإنك لا تخسر نقوداً ببناء الأشياء الخاطئة. في بعض الأحيان لا يكون المستخدمين متأكدين تماماً ما الذي يريدونه منذ البداية، وبذلك فقد تحتاج جهداً أكبر من المتوقع لتحديد ما الذي يريدونه فعلاً. ولكن بكافة الأحوال يبقى هذا أقل تكلفةً من بناء الشيء الخطأ ورميه بعيداً، ومن ثم البدء من جديد.

ومن المهم أيضاً أن تفكر بكيفية بناء النظام قبل البدء ببنائه. حيث أنه لا يجدر بك صرف الكثير من الوقت والمال ثم تذهب في النهاية إلى طريق مسدود عندما لا تكون مضطراً لذلك، خاصة عندما يؤدي ذلك إلى زيادة التكلفة.

العودة إلى القياس

إن بناء النظام البرمجي يشبه المشاريع الأخرى التي تحتاج للأشخاص والمال. أن تبني منزلاً، فذلك يعني أن تضع الرسومات المعمارية والمخططات قبل أن تبدأ بدق المسامير. ستراجع المخططات وتحسّنها قبل أن تصبّ الخرسانة. إن امتلاك الخطط الفنية مهم جداً في البرمجة.

فأنت لا تبدأ بتزيين شجرة عيد الميلاد قبل أن تضعها في مكانها، ولا تبدأ بإشعال النار حتى تفتح المدخنة، ولا تذهب في رحلة طويلة بخزان وقود فارغ، ولا تلبس ملابسك قبل أن تأخذ حماماً، كما أنك لا ترتدي حذاءك قبل جواربك. في البرمجة أيضاً يجب عليك فعل الأشياء بترتيبها الصحيح.

المبرمجون هم في نهاية السلسلة البرمجية. المُهيكِل (المسؤول عن الهيكلة) يعتمد على المتطلبات، والمصمم يعتمد على الهيكلة، وكاتب الشفرة يعتمد على التصميم.

بمقارنة السلسلة البرمجية بسلسلة التغذية الحقيقية. في وسط سليم بيئيا، طيور النورس تأكل سمك السلمون الطازج. وهو مغذي لها لأن السلمون أكل سمك الرنجة الطازج، التي بدورها أكلت بعوض الماء الطازج. النتيجة هي سلسلة غذائية صحيّة. في البرمجة، إن امتلكت تغذية صحية لكل مرحلة من مراحل سلسلة التغذية، فالنتيجة ستكون شفرة برمجية صحيّة مكتوبة من قبل مبرمجين سعداء.

أما في بيئة ملوثة، قد يسبح بعوض الماء في نفايات نووية، وسمك الرنجة يكون ملوث بثنائي الفينيل متعدد الكلور(PCBs)، والسلمون الذي يأكل الرنجة سبح خلال تسريبات النفط. طيور النورس تكون لسوء الحظ في نهاية سلسلة التغذية، وهكذا فأنها لا تأكل فقط النفط الذي في سمك السلمون السيئ. بل أيضاً تأكل ثنائي الفينيل متعدد الكلور والنفايات النووية من سمك الرنجة وبعوض الماء.

في البرمجة، إن كانت متطلباتك ملوثة، سوف تلوث الهيكلة، والهيكلة بدورها ستلوث البناء. هذا يقود إلى مبرمجين غاضبين سيئي التغذية، وبرمجيات ملوثة ومشعّة ومليئة بالعيوب.

إذا كنت تفكر بمشروع تكراري للغاية، ستحتاج لتحديد المتطلبات الهامة وعناصر الهيكلة التي تطبق على القطعة التي تبنيها قبل البدء بالبناء. البنّاء الذي يبني في مشروع التنمية السكنية لا يحتاج لأن يعرف كل تفصيل لكل منزل قبل البدء في بناء أول منزل. لكن البنّاء سيتفحص الموقع، ويخطط المجاري والخطوط الكهربائية، وإذا لم يجهّز البنّاء ذلك بشكل جيد، يمكن للبناء أن يتأخر في حال تطلبَ خط المجاري أن يُحفر تحت منزل قد تم بناءه بالفعل.

العودة الى للبيانات

أكدت الدراسات على مدى الـ 25 السنة الأخيرة بشكل قاطع على الدفع لفعل الأشياء الصحيحة من المرة الأولى. فالتغييرات غير الضرورية ستكون مكلفة.

بيانات مثبتة

وجد الباحثون في شركة هيوليت-باكارد(Hewlett-Packard)، و أي بي إم IBM، وشركة هيوز للطائرات (Hughes Aircraft، TRW، ومنظمات أخرى أن معالجة الخطأ في بداية البناء يسمح لإعادة العمل بأن يتم بكلفة أقل من 10 إلى 100 مرة عنه عندما ينجز في الجزء الأخير من العملية، أي خلال

اختبار النظام أو بعد إصداره (فاجان 1976- همفري، سنايدر، ويليس 1991- ليفنغويل 1997- ويليس وآخرون. 1998- غرادي 1999- شول وآخرون. 2002- بويم وتيرنز 2004).

في العموم، المبدأ هو إيجاد الخطأ في أقرب وقت ممكن لوقت تشكله. كلما طالت مدة بقاء العيب في سلسلة التغذية البرمجية، ستتسبب في أذى أكبر كلما انخفضنا في هذه السلسلة. بما أن المتطلبات تنجز أولاً، فإن عيوبها من المحتمل أن تبقى لمدة أطول في النظام وتصبح أكثر كلفة. العيوب التي تحصل في المراحل التمهيدية للبرنامج تميل أيضا إلى امتلاك تأثيرات أكبر من تلك التي تحصل في المراحل النهائية. هذا ما يجعل أيضاً من العيوب المبكرة أكثر تكلفة.

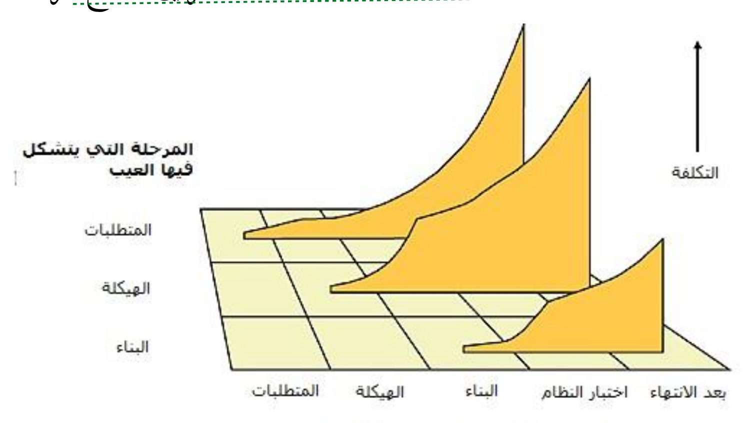
الجدول 3-1 يظهر التكلفة النسبية لتحديد العيوب بالاعتماد على وقت حدوثها ووقت اكتشافها.

بيانات مثبتة

وقت الإنتاج	وقت الاكتشاف				
	المتطلبات	الهيكلة	البناء	اختبار النظام	ما بعد الانتهاء
المتطلبات	1	3	5-10	10	10-100
الهيكلية	–	1	10	15	25-100
البناء	–	–	1	10	10-25

المصدر: تعديل من "تصميم وفحص الشفرة البرمجية للحد من الأخطاء في تطوير البرنامج" "Design and Code Inspections to Reduce Errors in Program Development" (فاغان 1976)، إزالة عيب البرمجيات " "Software Defect Remova" (دون 1984)،" تحسين عملية البرمجة في شركة هيوز للطائرات " " Software Process Improvement at Hughes Aircraft" (همفري، سنايدر، ويليس 1991)، "احتساب العائد من الاستثمار بإدارة متطلبات أكثر فعالية " " Calculating the Return on Investment from More Effective Requirements Management" (ليفنغويل 1997)،"تطوير شركة هيوز للطائرات لعملية تحسين البرمجيات باستمرار على نطاق واسع " " Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process" (ويلز وآخرون 1998)، "نموذج قرار الإصدار الاقتصادي: رؤى في إدارة مشروع البرمجيات "" An Economic Release Decision Model: Insights into Software Project Management" (غرادي 1999)، " ماذا تعلمنا من مكافحة العيوب"" What We Have Learned About Fighting Defects" (شول وآخرون 2002)، وموازنة السرعة والانضباط: دليل المتحير " Balancing Agility and Discipline: A Guide for the Perplexed" (بويم وتيرنز 2004).

كمثال، تظهر البيانات في الجدول 3-1 أن العيب الهيكلي الذي يكلف 1000دولار لمعالجته عند بدء تشكل الهيكلية يمكن أن يكلف 15000 دولار للمعالجة خلال اختبار النظام. الشكل 3-1 يوضح نفس الظاهرة.



الشكل 3-1 ترتفع تكلفة معالجة العيب إلى حد كبير مع زيادة الوقت بين تشكل العيب واكتشافه. يبقى هذا صحيحا سواء اكان المشروع متسلسلا بشكل كبير (القيام 100% من المتطلبات والتصميم قبل البدء) أو متكرر للغاية (القيام بـ5% من المتطلبات والتصميم قبل البدء).

بيانات مثبتة

لا يزال المشروع الوسطي يبذل معظم جهده في إصلاح العيوب على الجهة اليمنى من الشكل 3-1 الذي يعني أن الإصلاح وما يرتبط به من إعادة عمل يستغرق حوالي 50 بالمائة من وقت دورة تطوير البرمجيات النموذجية (ميلز 1983- بوهم 1987 - كوبر ومولن 1993 - فيشمان 1996- هالي 1996 - ويلر، بريكسزينسكي، وميسون 1996 – جونز 1998 – شل وآخرين 2002 – ويغرز2002). لقد وجدت عشرات الشركات أن التركيز ببساطة على تصحيح العيوب بوقت مبكر بدلاً من وقت متأخر في المشروع يمكن أن يقلل لمرتين أو أكثر من تكلفة التطوير والجدول الزمني للعوامل (ماكونيل McConnell 2004). وهذا حافز صحي لإيجاد ومعالجة مشاكلك بشكل مبكر قدر المستطاع.

اختبار الجاهزية الرئيس

عندما تعتقد أن رئيسك يفهم أهمية العمل على الشروط المسبقة قبل التحرك إلى البناء، جرب الاختبار التالي لتتأكد.

أي من هذه العبارات هي تنبؤات محققة ذاتياً؟

- من الأفضل أن نبدأ بكتابة الشفرة البرمجية حالاً لأنه سيكون لدينا العديد من التصحيحات للقيام بها.
- لم نخطط بوقت كافي للاختبار لأننا لن نجد عيوب كثيرة.
- لقد تحققنا من المتطلبات والتصميم كثيراً لدرجة أننا لا نستطيع التفكير بظهور أية مشاكل رئيسية خلال كتابة التعليمات البرمجية أو التنقيح.

كل هذه العبارات هي تنبؤات محققة ذاتياً، تهدف للوصول إلى أخر عبارة.

إن كنت لا تزال غير مقتنع بأن الشروط المسبقة تنطبق على مشروعك، القسم التالي سيساعدك بأن تأخذ قرارك.

2.3 تحديد نوع البرمجيات التي تعمل عليها

لخّص كابيرز جونز، كبير العلماء في بحوث إنتاجية البرمجيات، 20 سنة من البحث البرمجي بالإشارة إلى أنه هو وزملائه قد شهدوا 40 طريقة مختلفة لجمع المتطلبات، و50 اختلاف في العمل على تصاميم البرمجيات، و30 نوع من الاختبارات المطبقة على المشروع في أكثر من 700 لغة برمجة مختلفة (جونز 2003).

أنواع مختلفة من مشاريع البرمجة تدعوا لموازنة مختلفة بين التحضير والبناء. كل مشروع يكون فريداً من نوعه، لكن المشاريع تميل للوقوع في نمط التطويرات العامة. الجدول 3-2 يظهر ثلاثة من أكثر أنواع المشاريع الشائعة ويعرض عادة الأنشطة الأنسب لكل نوع من المشاريع.

الجدول 3-2 الأنشطة القياسية الجيدة لثلاثة أنواع شائعة من المشاريع البرمجية

	نوع البرمجيات		
	أنظمة العمل	**أنظمة المهام الحرجة**	**أنظمة الحياة الحرجة المضمَنة**
تطبيقات نموذجية	موقع انترنت موقع انترانت إدارة المخزون الألعاب نظم المعلومات الإدارية أنظمة الرواتب	برمجيات مضمَنة الألعاب موقع انترنت البرمجيات المحزَمة أدوات البرمجيات خدمات الويب	برمجيات الطيران الالكترونية برمجيات مدمجة أجهزة طبية أنظمة التشغيل البرمجيات المحزمة
نماذج دورة الحياة	التطوير المرن (البرمجة الفائقة، سكروم Scrum[1]، تطوير الإطار الزمني وغيرها) التطور النماذج الأولية	التسليم على مراحل التطور التسليم التطوير المتصاعد(اللولبي)	التسليم على مراحل التطوير المتزايد التسليم التدريجي (التطوري)
التخطيط والإدارة	تخطيط المشاريع التزايدي الاختبار عند الحاجة وتخطيط ضمان الجودة التحكم بالتغيرات غير النظامية	التخطيط الأساسي الأولي تخطيط الاختبار الأساسي الاختبار عند الحاجة وتخطيط ضمان الجودة التحكم بالتغيرات النظامية	التخطيط الشامل الأولي تخطيط الاختبار الشامل تخطيط ضمان الجودة الشامل التحكم بالتغيرات الدقيقة
المتطلبات	مواصفات المتطلبات غير النظامية	مواصفات المتطلبات شبه النظامية مراجعة المتطلبات عند الحاجة	مواصفات المتطلبات النظامية الفحص للمتطلبات النظامية
التصميم	تم دمج التصميم وكتابة الشفرة البرمجية	تصميم الهيكلة	تصميم الهيكلة الفحص الرسمي للهيكلة

[1] سكرَمْ هو أحد إطارات العمل وفقاً لمقاييس منهجية تطوير البرمجيات أجايل لإدارة تطوير المنتجات. يتميز بأنه ذو نمط تكراري وتزايدي.

		التصميم التفصيلي غير الرسمي مراجعات للتصميم عند الحاجة	التصميم التفصيلي الرسمي الفحص غير الرسمي للتصميم التفصيلي
البناء	البرمجة الثنائية أو كتابة الشفرة الفردية إجراءات الكشف غير الدقيقة أو عدم وجود إجراءات الكشف	البرمجة الثنائية أو كتابة الشفرة الفردية إجراءات الكشف غير الدقيقة مراجعات الشفرة عند الحاجة	البرمجة الثنائية أو كتابة الشفرة الفردية إجراءات الكشف الدقيقة الفحص الرسمي للشفرة
الاختبار وضمان الجودة	المطورين يختبرون شفرتهم الخاصة الاختبار بداية التطوير بدون أي اختبارات أو القليل من الاختبارات من قبل مجموعة اختبار مستقلة	المطورين يختبرون شفرتهم الخاصة الاختبار بداية التطوير مجموعة اختبار مستقلة	المطورين يختبرون شفرتهم الخاصة الاختبار بداية التطوير مجموعة اختبار مستقلة مجموعة ضمان جودة مستقلة
التطوير	إجراءات التطوير غير الرسمية	إجراءات التطوير الرسمية	إجراءات التطوير الرسمية

في المشاريع الحقيقية، ستجد اختلافات غير منتهية للمواضيع الثلاثة المقدَّمة في هذا الجدول، على أية حال، سلطنا الضوء في الجدول على العموميات. تتجه مشاريع أنظمة العمل إلى الاستفادة من النُهج الشديدة التكرارية، والتي يوجد مسافة بين تخطيطها، ومتطلباتها، وهيكلتها وبين بنائها، واختبار نظامها، ونشاطات ضمان الجودة. وتتجه نظم الحياة الحساسة إلى تطّلب نُهج أكثر تعاقبية. استقرار المتطلبات هو جزء من الحاجات التي تضمن مستويات عالية جداً من الموثوقية.

تأثير النُهج التكرارية على المتطلبات المسبقة

أكدّ بعض الكتّاب أنّ المشاريع التي تستخدم تقنيات تكرارية لا تحتاج إلى التركيز على المتطلبات مطلقاً، لكن قُدمت وجهة النظر هذه بشكل خاطئ. فالنُهج التكرارية تسعى لتخفيض الأثر الناتج عن الأعمال التحضيرية غير الملائمة، لكن لا تلغيه. انظر الى الأمثلة الموجود في الجدول 3-3 لمشاريع لم تركز على المتطلبات. أنجز مشروع منها بشكل تعاقبي واعتمد على الاختبار فقط لاكتشاف الخلل؛ المشروع الآخر أُنجز بشكل تكراري وكان يتم اكتشاف الخلل خلال تقدم المشروع. أخّرَ النهج الأول القسم الأكبر من تصحيح الأخطاء الى نهاية المشروع، جاعلاً الكلفة أكبر، كما دُوّن في الجدول 3-1. النهج التكراري يمتص إعادة العمل تدريجياً خلال مسار المشروع، مما يجعل الكلفة الكليّة أقل. البيانات في هذا الجدول والجدول التالي لغرض التوضيح فقط، لكن الكُلف الموافقة للنهجين العامين قُدمت بشكل جيد في البحث الموصوف سابقاً في هذا الفصل.

الجدول 3-3 تأثير تجاوز المتطلبات في المشاريع التكرارية والتعاقبية

	النهج #1: نهج تعاقبي بدون متطلبات		النهج #2: نهج تكراري بدون متطلبات	
إكمال المشروع	كلفة العمل	كلفة إعادة العمل	كلفة العمل	كلفة إعادة العمل
20%	100000$	0$	100000$	75000$
40%	100000$	0$	100000$	75000$
60%	100000$	0$	100000$	75000$
80%	100000$	0$	100000$	75000$
100%	100000$	0$	100000$	75000$
إعادة العمل في نهاية المشروع		500000$		0$
المجموع	500000$	500000$	500000$	375000$
المجموع الكلي		1000000$		875000$

يختلف المشروع التكراري الذي يختصر أو يلغي المتطلبات بأمرين عن المشروع التعاقبي الذي يقوم بنفس العمل. أولاً، سيصبح متوسط كُلف تصحيح الخطأ أقل لأن الأخطاء ستصبح أكثر قابلية للاكتشاف بوقت قريب الى وقت الذي حُشرت به في البرمجية. على كل حال، الأخطاء ستبقى تُكتشف لاحقاً في كل كرّة، وتصحيحها يتطلب إعادة (تصميم، وكتابة شفره، واختبار) لأجزاء معينة من البرمجية، ما يجعل كلف تصحيح الأخطاء أعلى مما تحتاجه. ثانياً، مع النُهج التكرارية سيتم امتصاص الكلف تدريجياً، خلال المشروع، بدلاً من تعاظمها في نهاية المشروع. بعد أن ينتهي المشروع، الكلف الكليّة ستكون متقاربة لكن لم تكن لتبدو بهذا الفارق لان السعر يُدفع على أقساط صغيرة خلال مسار المشروع، بدلاً من أن يُدفع كلّه مرة واحدة في النهاية.

كما يوضح الجدول 3-4، يمكن للتركيز على المتطلبات أن يخفض الكلف بغض النظر عن النهج المختار التكراري أو التعاقبي. النُهج التكرارية عادةً خيار أفضل لأسباب عدّة، لكن يمكن أن ينتهي النهج التكراري الذي تجاهل المتطلبات بكلفة أعلى بشكل ملحوظ عن المشروع التعاقبي الذي أعطى انتباهاً جيداً للمتطلبات.

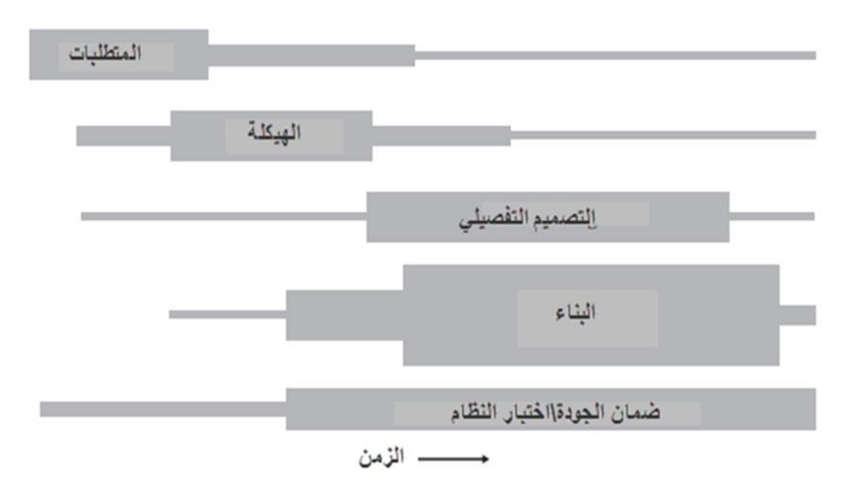
الجدول 3-4 تأثير التركيز على المتطلبات في المشاريع التكرارية والتعاقبية

	النهج #3: نهج تعاقبي مع متطلبات		النهج #4: نهج تكراري مع متطلبات	
إكمال المشروع	كلفة العمل	كلفة إعادة العمل	كلفة العمل	كلفة إعادة العمل
20%	100000$	20$	10$	75000$
40%	100000$	20$	10$	75000$
60%	100000$	20$	10$	75000$
80%	100000$	20$	10$	75000$
100%	100000$	20$	10$	75000$
إعادة العمل في نهاية المشروع		0$		0$
المجموع	500000$	100000$	500000$	50000$
المجموع الكلي		600000$		50000$



كما يقترح الجدول 3-4 أغلب المشاريع ليست تعاقبية أو تكرارية بشكل كامل، وليس من العملية تحديد 100 بالمئة من المتطلبات أو التصميم قبل البدء؛ ولكن تجد أغلب المشاريع فائدة في تعريف -على الأقل- أهم المتطلبات والعناصر الهيكلية حساسية في وقت مبكر.

وأحد القواعد الشائعة وهي التخطيط لتحديد حوالي 80 بالمائة من المتطلبات في البدء، وتخصيص وقت للمتطلبات الإضافية ليتم تحديدها لاحقا[1]، ثم مارس نظام تحكم ممنهج للتغيرات لقبول المتطلبات الجديدة الأكثر قيمة مع مضي المشروع، بديل أخر هو تحديد أهم 20 بالمائة من المتطلبات قبل البدء فقط، وخطط لتطوير باقي البرنامج بزيادات بسيطة؛ مع تحديد المتطلبات والتصاميم الإضافية كلما تقدمت. يعكس الشكلان 3-2 و3-3 هذه النُهج المختلفة.



الشكل 3-2 **الأنشطة تتداخل إلى مستوى معين في معظم المشروع، حتى تلك المتعاقبة بشكل جليّ**

[1] **إشارة مرجعية** لتفاصيل عن كيفية ملائمة نهجك في تطوير برامج بحجوم مختلفة راجع الفصل 27، "كيف يؤثر حجم البرنامج على البناء"



الشكل 3-3 في مشاريع أخرى، الأنشطة تتداخل خلال مدة المشروع. أحد الأساسيات لبناء ناجح هو معرفة مستوى كمال المتطلبات وضبط النهج تبعاً لها.

اختيار النُهج التكرارية أو المتعاقبة

يتفاوت مدى استيفاء الشروط المسبقة بوضوح قبل البدء مع نوع المشروع المبين في الجدول 3-2، وشكل المشروع، والبيئة التقنية، ومقدرات الطاقم، وأهداف المشروع التجارية.

اختر نُهج بوضوح أكثر تعاقبية عندما:

- المتطلبات مستقرة تماماً.
- التصميم مستقيم ومفهوم تماماُ بشكل جيد.
- فريق التطوير يألف مجال تطبيق المشروع.
- يحوي المشروع مخاطرة صغيرة.
- يكون التنبؤ لزمن بعيد مهم.
- تكون كلفة تغيير المتطلبات، والتصميم، والشفرة مع السياق (code downstream) على الأرجح عالية.

اختر نهج أكثر تكرارية (كلما تقدمت) عندما:

- تكون المتطلبات غير مفهومة بشكل جيد، أو عندما تتوقع أن تكون المتطلبات غير مستقرة لأسباب أخرى.
- يكون التصميم معقد أو فيه تحدّي أو كليهما.
- لا يألف فريق التطوير مجال تطبيقات المشروع.

- يحوي المشروع الكثير من المخاطر.
- يكون التوقع لزمن بعيد غير مهم.
- تكون كلفة تغيير المتطلبات، والتصميم، والشفرة مع السياق على الأرجح منخفضة.

كون البرمجيات على ما هي عليه، فإن النُهج التكرارية تكون أكثر فائدة من النُهج التعاقبية. تستطيع أن تكيّف المتطلبات مع مشروعك المحدد عن طريق جعلها أكثر أو أقل رسميةً، وأكثر أو أقل كمالاً، كما تراه مناسباً. الق نظرة على الفصل 27 لنقاش مفصّل عن النُهج المختلفة لمشاريع كبيرة وصغيرة (تُعرف أيضاُ بالنُهج المختلفة للمشاريع الرسمية وغير الرسمية).

إليك التصور الصافي حول متطلبات البناء: عليك في البداية أن تحدد متطلبات البناء المناسبة بجودة لمشروعك. في بعض المشاريع يُصرف وقت قليل على المتطلبات، ما يعرّض البناء بشكل غير ضروري إلى معدلٍ عالٍ لتغيرات تسبب عدم استقرار المشروع، ويمنع المشروع من التقدم بشكل ثابت. وبعض المشاريع تقوم بعمل الكثير من الايضاح، وتلتزم بحزم بالمتطلبات والخطط التي أُبطلت بالاكتشافات التصريفية، وهذا يمكن أيضاً أن يعيق التقدم خلال البناء.

الآن، وبعد أن درست الجدول 3-2 وحددت المتطلبات المناسبة لمشروعك، يصف الجزء المتبقي من هذا الفصل كيفية تحديد ما إذا كان كل متطلب بناء سيضم لقائمة المتطلبات أو قائمة الحذف.

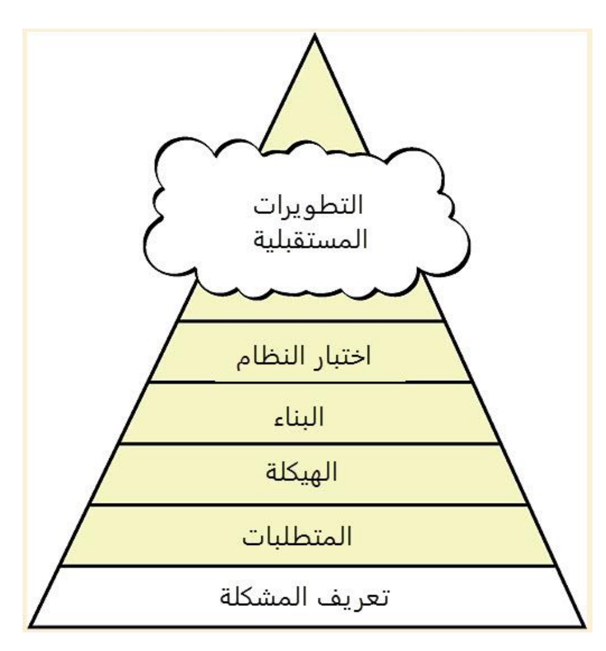
3.3 متطلبات تعريف المشكلة

أول متطلب عليك التحقق منه قبل البدء بالبناء هو تقرير واضح عن المشكلة التي يُفترض بالنظام أن يحلّها.[1] هذا ما يدعى أحياناً ب "رؤية المنتج" أو "تقرير الرؤية" أو "تقرير المهمة" أو "تعريف المنتج" هنا يدعى "تعريف المشكلة." طالما أن هذا الكتاب عن البناء، لن يخبرك هذا القسم كيف تكتب تعريف المشكلة؛ على كلٍ، سيخبرك كيف تتعرف إن كُتب تعريف وإن كان هذا الذي كُتب سيشكل أساساً جيداً للبناء.

[1] إذا كان "الصندوق" هو حدود تقع ضمنها الشروط والقيود، إذن البراعة أن تجد الصندوق... لا تفكر خارج الصندوق -جِد الصندوق! -أندي هنت وديف توماس

يعرّف تعريف المشكلة ما هي المشكلة دون الاشارة الى أي حلّ محتمل. إنّها تقرير بسيط ربما صفحة أو صفحتين، وعليها أن تبدو كمشكلة. العبارة " لا نستطيع تلبية طلبات غيغاترون" تبدو كمشكلة، وهي تعريف جيد للمشكلة. العبارة "نحتاج أن نحسن نظام إدخال البيانات الآلي كي نتمكن من تلبية طلبات غيغاترون" هي تعريف فقير للمشكلة، إنّها لا تبدو كمشكلة لكن تبدو كحل.

كما يبدو في الشكل 3-4، تعريف المشكلة يأتي قبل عمل المتطلبات التفصيلي، والذي يعتبر استقصاء أكثر عمقاً عن المشكلة.



الشكل 3-4 تعريف المشكلة يشكل الاساس للباقي من عملية البرمجة

ينبغي أن يكون تعريف المشكلة بلغة المستخدم، وينبغي أن توصف المشكلة من وجهة نظر المستخدم. لا ينبغي عادةً أن تصاغ بمصطلحات الحاسوب التقنية. أفضل الحلول ربما ليس برنامج حاسوبي. افترض أنّك تريد تقرير يعرض الأرباح السنوية. ولديك مسبقاً تقارير حاسوبية تعرض الأرباح الربعية. إذا كنت محبوس داخل آلية التفكير البرمجية، ستفكر أن إضافة تقرير سنوي لنظام لديه مسبقاً تقارير ربعية أمر سهل. بعدها ستدفع لمبرمج كي يكتب ويختبر برنامج يأخذ الكثير من الوقت لحساب الأرباح السنوية. إذا لم تكن محبوس داخل آلية التفكير البرمجية تلك، ستتدفع لسكرتيرك كي ينشئ التقرير السنوي بدقيقة يجمع خلالها الأرباح الربعية بحاسبة جيبية.

الاستثناء لهذه القاعدة عندما تكون المشكلة تخصّ الحاسوب: وقت الترجمة بطيء أو أدوات البرمجة بالية. عندها من المناسب أن تصوغ المشكلة بمصطلحات البرمجة أو الحاسوب. كما يوضح الشكل 3-5، بدون تعريف جيد للمشكلة، ربما ستبذل جهداً لحل المشكلة الخطأ.



الشكل 3-5 تأكد علامَ تصوب قبل أن تطلق



ضريبة الإخفاق في تعريف المشكلة أنّه يمكن أن تضيّع الكثير من الوقت في حل المشكلة الخطأ. وهي ضريبة مزدوجة لأنّك حتى لم تحلّ المشكلة المراد حلّها.

3.4 متطلبات الاحتياجات الأولية

المتطلبات تصف بتفصيل ما على النظام القيام به، وهي الخطوة الأولى باتجاه الحلّ. ويعرف نشاط المتطلبات أيضاً بـ "تطوير المتطلبات" "تحليل المتطلبات" "التحليل" "تعريف المتطلبات" "متطلبات البرمجية" "التوصيف" "التوصيف الوظيفي".

لماذا متطلبات رسمية؟

وجود مجموعة واضحة من المتطلبات مهم لعدة أسباب.

تساعد المتطلبات الصريحة في ضمان توجيه المستخدم لوظائف النظام بدلاً من المبرمج. إذا كانت المتطلبات واضحة، يستطيع المستخدم مراجعتها وقبولها. وإذا لم تكن، ينتهي الحال بالمبرمج باتخاذ قرارات بخصوص المتطلبات خلال البرمجة. المتطلبات الواضحة تحفظك من تخمين ما يريد المستخدم.

تساعد أيضاً المتطلبات الصريحة في اجتناب الجدل. أنت كمبرمج تتخذ قرارات في مجال النظام قبل البدء بالبرمجة. فإذا لم تتفق مع مبرمج آخر حول ما يُفترض على النظام القيام به، بإمكانك إن تحلّ ذلك بالرجوع إلى المتطلبات المكتوبة.



إعطاء الانتباه إلى المتطلبات يساعد في تحجيم التغيرات التي يمكن أن يتعرض لها النظام بعد البدء بالتطوير. إذا وجدت خطأ في الشفرة خلال كتابة الشفرة البرمجية، تُغيّر أسطر قليلة من الشفرة وينتهي الأمر. أما إذا وجدت خطأ في المتطلبات خلال كتابة الشفرة، عليك أن تبدّل التصميم ليوافق المتطلبات الجديدة. ربما ستضطر إلى رمي جزء من التصميم القديم، ولأن التصميم الجديد عليه أن يتأقلم مع الشفرة

الموجودة، سيأخذ التصميم وقت أطول مما لو بدأت به من الصفر. وعليك أيضاً أن تهمل الشفرة البرمجية وحالات الاختبار التي تأثرت بتغيير المتطلبات وكتابة الشفرة وحالات اختبار جديدة. حتى الشفرة التي لم تتأثر يجب ولا بد أن يعاد اختبارها كي تضمن أن التغيرات لم تنتج أخطاء جديدة في مناطق أخرى.

بيانات مثبتة

كما أظهر الجدول 3-1، حددت بيانات من عدة منظمات أنّ خطأ المتطلبات في المشاريع الكبيرة إذا اكتُشف خلال مرحلة الهيكلة سيكون التصحيح بشكل قياسي أغلى 3 مرات منه في حال اكتُشف في مرحلة المتطلبات. وإذا اكتشف خلال كتابة الشفرة، سيكون أغلى 5-10 مرات؛ خلال اختبار النظام، 10 مرات، وبعد إطلاق المنتج، يا للهول، 10-100 مرة من كلفة التصحيح إذا اكتشف خلال تطوير المتطلبات. في المشاريع الصغيرة وبكلف إدارية أخفض، معامل الضرب في حالة بعد الاطلاق أقرب الى 5-10 منه عن 100 (بويم وتُرنير 2004). في كلتا الحالتين، ليس المال هو الشي الذي تريد أن يُأخذ من راتبك.

تحديد المتطلبات بكفاية أساس لمشروع ناجح، وربما أهم حتى من تقنيات البناء الفعّال. (انظر الشكل 3-6.) كُتبت العديد من الكتب عن كيفية تحديد المتطلبات بجودة. وهكذا، لا تخبرك الأقسام القليلة التالية كيف تقوم بعمل جيد في تحديد المتطلبات، لكن تخبرك كيف تحدد إذا كانت المتطلبات قد أُنجزت بشكل جيد وكيف تصنع الأفضل من المتطلبات التي بين يديك.



الشكل 3-6 بدون متطلبات جيدة، ستستطيع أن تعرف المشكلة العامة المناسبة لكن ستضيع الهدف في مفاهيم محددة من المشكلة.

أسطورة المتطلبات المستقرة

المتطلبات المستقرة هي الكأس المقدسة في تطوير البرمجيات.[1] مع متطلبات مستقرة، يمكن أن يتقدم المشروع من الهيكلة إلى التصميم إلى الشفرة الى الاختبار بطريقة مرتبة، ومتوقعة، وسلسة. هذه هي جنة

[1] المتطلبات كالماء. البناء عليها وهي متجمدة أسهل. -مجهول

البرمجيات! لديك نفقات متوقَّعة، وليس عليك أن تقلق بشأن الميزة التي تكلف 100 مرة أكثر كي تتحقق مما لو كانت غير ذلك لأنّ المستخدم لم يفكر بها حتى انتهيت أنت من التصحيح (debugging).

من الجيد أن تأمل عدم الحاجة الى أية تغييرات بعد موافقة الزبون على المتطلبات. على أية حال، في المشاريع القياسية لا يستطيع الزبون أن يصف ما يحتاج إليه بموثوقية قبل كتابة الشفرة. ليست المشكلة أن الزبائن قليلي المعرفة بهذا المجال، فعلى قدر عملك في المشروع يكون فهمك أفضل له، وعلى قدر ما يعملون به يكون فهمهم أفضل له. تساعد عملية التطوير الزبائن في فهم حاجاتهم أكثر، وهذا السبب الاساسي لتغيّر المتطلبات (كورتيس، كراسنير، إيسكو 1099، جونز 1009، ويغيرز 2003). خطة تتبع المتطلبات بصلابة هي في الحقيقة خطة أن لا تستجيب لزبونك.

إلى أي مدى يكون التغيّر أمر اعتيادي؟ وجدت دراسات في "أي بي إم" وشركات أخرى أنّ متوسط المشاريع يتعرض لحوالي 25% من التغيّر في المتطلبات خلال التطوير (بويم 1981، جونز 1994، جونز 2000)، والذي يؤدي الى 70 -85 بالمئة من إعادة العمل في المشروع القياسي (ليفينغويل 1997، ويغيرز 2003).

ربما تفكر أن "بونتياك أزتيك" كانت أفضل سيارة صنعت على الإطلاق، وفقاً لمجتمع الأرض المسطحة، وتقوم بالحج إلى مهبط الكائنات الفضائية في المكسيك-روسويل، كل أربع سنوات. إذا كنت تفعل ذلك، تشبث برأيك وصدّق أن المتطلبات لن تتغير في المشاريع. من الناحية الأخرى، إذا توقفت عن الإيمان ببابا نويل وجنيّة الأسنان، أو على الأقل توقفت عن الاعتراف بهم، بإمكانك أن تأخذ عدة خطوات لتحجّم أثر تغيّر المتطلبات.

التعامل مع التغيّر في المتطلبات خلال البناء



هاهنا عدة أشياء يمكن أن تفعلها لتصنع الأفضل من تغيّر المتطلبات خلال البناء:

استخدم قائمة المتطلبات في نهاية هذا القسم لتقدر جودة المتطلبات لديك إذا لم تكن المتطلبات جيدة كفاية، توقف عن العمل، تثبّت، اجعل المتطلبات صحيحة قبل المتابعة. بالتأكيد، ستشعر وكأنك ترجع إلى الوراء إذا توقفت عن الشفرة في هذه المرحلة. لكن إذا كنت تسوق من شيكاغو إلى لوس أنجلوس، هل هي مضيعة للوقت أن تتوقف وتنظر إلى خارطة الطريق إذا رأيت لافتات لنيويورك؟ لا. إذا لم تكن متوجهاً في الاتجاه الصحيح، توقف وتفحص المسار.

تأكد من أنّ الجميع يعلمون كلفة تغيّر المتطلبات يتحمس الزبائن عندما يفكرون بميزة جديدة. بحماسهم، يخف دمهم ويتجه إلى بصلتهم السيسائية ما يجعلهم يدوخون، ناسين كل المقابلات التي قمت بها لتناقش

المتطلبات، وحفل التوقيع، ووثيقة المتطلبات التامة. الطريقة الأسهل لتتعامل مع هكذا أناس مخمورين بالميزات هي أن تقول، "امممممم، إنها تبدو فكرة عظيمة. وطالما أنها ليست في وثيقة المتطلبات، سأقوم بإنشاء جدول أعمال منقح وتخمين للكلفة بحيث تستطيع أن تقرر إذا كنت تريد أن نقوم بها الآن أو لاحقاً." الكلمات "جدول أعمال" و"كلفة" أشدّ وطئا من القهوة وحمام بارد، الكثير من " حتماً يجب أن يكون لدينا" ستتحول بسرعة الى "شيء لطيف أن يكون لدينا"

إذا كانت منظمتك غير مبالية بأهمية القيام بالمتطلبات أولاً، نبه إلى أن التغيّرات في فترة المتطلبات أرخص من التغيّرات لاحقاً. استخدم من هذا الفصل نقاشات مقنعة تماماً وسهلة جداً للقيام بالمستلزمات قبل البناء.

أنشئ آلية للتحكم بالتغيّرات[1] إذا كان زبائنك ثابتون على الحماس، خذ بعين الاعتبار تأسيس لوحة التحكم بالتغيرات الرسمية لمراجعة مثل هكذا تغيرات مقترحة. إنه أمر جيد أن يغير الزبائن آراءهم ويتحققوا من أنهم يحتاجون مقدرات أكثر. ولكن المشكلة هي أن تغيراتهم المقترحة متكررة جداً لدرجة أنك لا تستطيع تلبيتها. إن وجود آلية مدمجة للتحكم بالتغيرات يجعل الكل سعيداً. ستكون سعيداً لمعرفتك بأنك ستعمل مع التغيرات لمرات محددة. وزبائنك سيكونون سعداء لمعرفتهم بأنه لديك مخطط للتعامل مع مساهماتهم.

استخدم نُهج التطوير التي تتكيف مع التغيرات[2] بعض نُهج التطوير تعطيك أفضل مقدرة للتعامل مع المتطلبات المتغيرة. نهج نموذج بدائي تطوري يساعد في اكتشاف متطلبات النظام قبل أن ترسل قواتك للبناء. التسليم التطوري هو نهج يسلم النظام على مراحل. بإمكانك أن تبني قليلاً، وتحصل على القليل من التغذية الراجعة من المستخدمين، وتعاير تصميمك قليلاً، وتصنع القليل من التغيرات، وتبني قليلاً مجدداً. المفتاح هنا هو استخدام دورات تطوير قصيرة بحيث يمكنك الاستجابة لزبائنك بسرعة.

اترك المشروع[3] إذا كانت المتطلبات سيئة بشكل كبير أو متقلبة ولم تجد أي من الاقتراحات المذكورة أعلاه، الغ المشروع. حتى إذا لم تستطع إلغاء المشروع، فكّر بالذي يجب كي تلغيه. فكّر بكم السوء الذي ستكون عليه إذا لم تلغيه. إن وجدت حالة ستهمل المشروع فيها، على الأقل اسأل نفسك كم هو الفرق بينها وبين حالتك.

[1] **إشارة مرجعية** لتفاصيل حول التعامل مع التغيرات في التصميم والشفرة، انظر القسم 28.2، "إدارة الترتيبات"

[2] **إشارة مرجعية** لتفاصيل حول نُهج التطوير التكراري، انظر "التكرار" في القسم 5.4 والقسم 29.3، "استراتيجيات التكامل المتزايد"

[3] **للاطلاع** لتفاصيل عن نُهج التطوير التي تدعم المتطلبات المرنة، راجع Rapid Development (McConnell 1996)

ركز اهتمامك على قضية عمل مشروعك[1] الكثير من قضايا المتطلبات تختفي أمام عينيك عندما ترجع الى سبب العمل في مشروعك. المتطلبات التي بدت كفكرة جيدة عندما اعتبرت كـ "ميزة" يمكن أن تبدو كفكرة مزعجة عندما تقيّم "قيمة العمل التراكمي." المبرمجون الذين يتذكرون أن يأخذوا بعين الاعتبار تأثير قراراتهم على العمل يساوون بثقلهم ذهباً- على الرغم من أنني سأكون سعيداً إذا تلقيت ثمن هذه النصيحة بالمال.

لائحة المهام: المتطلبات[2]

تحوي لائحة مهام المتطلبات مجموعة من الأسئلة لتسألها لنفسك عن متطلبات مشروعك. لا يخبرك هذا الكتاب كيف تقوم بتطوير متطلبات جيد، ولن تخبرك اللائحة أيضاً. استخدم اللائحة كفحص للسلامة العقلية في مدة البناء لتحدد كم هي صلبة الأرض التي تقف عليها- أين تكون على مقياس رختر للمتطلبات.

لن تُطبَّق كل الأسئلة على مشروعك. إذا كنت تعمل في مشروع غير رسمي، ستجد بعضها حتى لا يستدعي أن تفكر به. ستجد بعضها الآخر يستدعي أن تفكر به لكن لا يستدعي أن تجاوب عليه رسمياً. على أية حال، إذا كنت تعمل في مشروع ضخم ورسمي ربما عليك أن تأخذ بعين الاعتبار كل سؤال.

المتطلبات الوظيفية المتخصصة

- هل كل مدخلات النظام محددة ومتضمنة مصدرها، ودقتها، ومجال قيمها، وتكرارها؟
- هل كل مخرجات النظام محددة ومتضمنة هدفها، ودقتها، ومجال قيمها، وتكرارها؟
- هل كل صيغ المخرجات محددة لصفحات الوب، والتقارير، وما أشبه؟
- هل كل الواجهات البرمجية والصلبة الإضافية محددة؟
- هل كل واجهات الاتصال الاضافية محددة، ومتضمنة "التصافح"، وتفحص الأخطاء، وبروتوكولات الاتصال؟
- هل كل المهام التي يريدها المستخدم أن تنجز محددة؟
- هل البيانات المستخدمة في كل مهمة والبيانات الناتجة عن كل مهمة محددة؟

المتطلبات المحددة غير الوظيفية (متطلبات الجودة)

- هل زمن الاستجابة المتوقع، من وجهة نظر المستخدم، محددة لكل العمليات الضرورية؟
- هل الاعتبارات الزمنية الأخرى محددة، مثل وقت المعالجة، ومعدل نقل البيانات، وإنتاجية النظام؟
- هل درجة الأمان محددة؟

[1] **إشارة مرجعية** لتفاصيل عن الفرق بين المشاريع الرسمية وغير الرسمية (المسببة بالغالب من الفرق بحجم المشروع)، راجع الفصل 27، "كيف يؤثر حجم البرنامج على البناء"

[2] cc2e.com/0323

- هل الموثوقية محددة، ومتضـــمنة تبعات فشـــل البرمجية، والمعلومات الحيوية التي يجب أن تُحمى من الخلل، واستراتيجية اكتشاف الاخطاء والاسترداد؟
- هل الحد الأدنى لذاكرة الآلة والمساحة الحرة محددة؟
- هل إمكانية صيانة النظام محددة، متضمنة قدرته على التكيف مع متغيرات في وظيفة محددة، والتغيّر في بيئة العمل، والتغير في واجهاته مع برمجيات أخرى؟
- هل تعريف النجاح مضمن؟ وكذلك الفشل؟

متطلبات الجودة

- هل كتبت المتطلبات بلغة المستخدم؟ وهل يعتقد المستخدمون ذلك؟
- هل كل متطلب يتجنب التصادم والتعارض مع المتطلبات الأخرى؟
- هل بالإمكان إجراء مقايضـات مقبولة بين السـمات المتنافسـة المحددة، على سـبيل المثال، بين المتانة والصحة؟
- هل تتجنب المتطلبات تحديد التصميم؟
- هل تحوي المتطلبات على مستوى وا ضح من التفا صيل؟ أيجب أن يحدد أي متطلب بتفا صيل كثيرة؟ أيجب أن يحدد أي متطلب بتفاصيل أقل؟
- هل المتطلبات واضـــحة بما فيه الكفاية ليتم تســـليمها إلى مجموعة مســـتقلة للبناء وتبقى مفهومة؟ هل يعتقد المطورون ذلك؟
- هل كل بند يتعلق بالمشكلة وحلها؟ هل يمكن أن يعزى كل بند إلى أصله في بيئة المشكلة؟
- هل كل متطلب قابل للاختبار؟ وهل سيكون من الممكن اجراء اختبار مستقل لتحديد ما إذا كان قد تم استيفاء كل المتطلبات؟
- هل حُددت جميع التغييرات الممكنة للمتطلبات، بما في ذلك احتمال حدوث كل تغيير؟

كمال المتطلبات

- عندما تكون المعلومات غير متوفرة قبل أن يبدء التطوير، هل تم تحديد مناطق عدم الاكتمال؟
- هل المتطلبات كاملة بمعنى أنه إذا كان المنتج يستوفي كل متطلب، سوف يكون مقبولا؟
- هل أنت مرتاح لجميع المتطلبات؟ هل قمت بإقصـــاء المتطلبات التي من المســـتحيل تنفيذها وضمنّتها فقط لاسترضاء عميلك أو رئيسك في العمل؟

3-5 متطلبات الهيكلة الأولية[1]:

[1] **إشارة مرجعية** لمزيد من التفاصيل عن تصميم البرامج ذات المستوى الأدنى، انظر الفصول من 5 حتى 9

هيكلة البرمجيات هي جزء عالي المستوى من تصميم البرمجيات، والإطار الذي يجمع الأجزاء الأكثر تفصيلاً للتصميم (بوشمان وآخرون 1996، فاولر 2002، باس كليمنتس كازمان 2003، كليمنتس وآخرون2003)

ويعرف بناء البرمجيات كـ " بناء النظام " و" تصميم المستوى الأعلى " وأعلى مستوى للتصميم". توصف هيكلة البرمجيات نموذجياً بمستند واحد يشار اليه كـ (مواصفات الهيكلة) أو " تصميم المستوى الأعلى ".

يفرّق بعض النّاس بين الهيكلة والتصميم عالي المستوى_ حيث تشير هيكلة البرمجيات إلى تصميم قيود تطبق على النظام الشامل بينما يشير التصميم عالي المستوى إلى تصميم قيود تطبق على النظام الفرعي أو على مستوى الصفوف المتعددة، وليس بالضرورة على النظام الشامل.

ولأن هذا الكتاب هو حول البناء، فهذا المقطع لا يعلمك كيفية تطوير هيكلة البرمجيات، إنه يركز على كيفية تحديد كفاءة هيكلة برمجية موجودة مسبقاً.

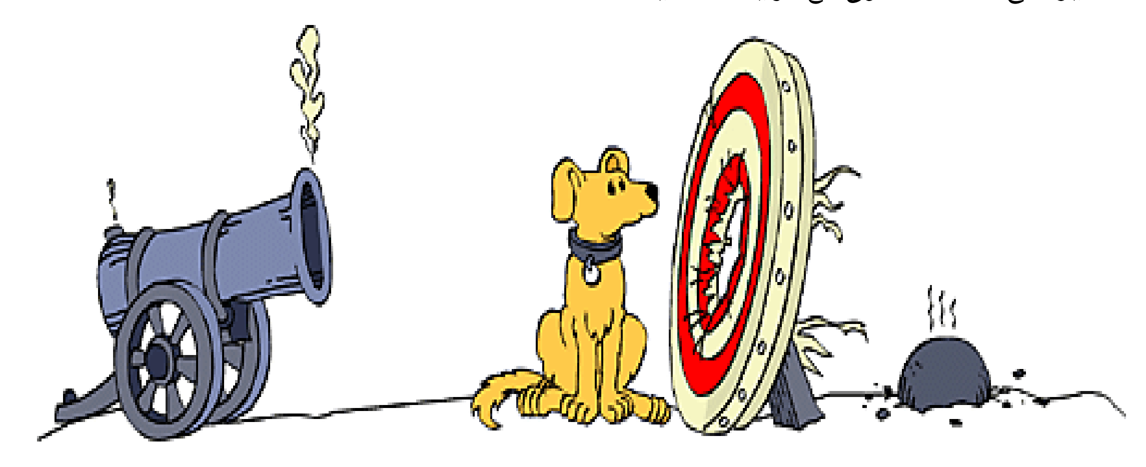
ولأن الهيكلة أقرب بخطوة واحدة للبناء منها للمتطلبات، على كل حال، فإن مناقشة هيكلة البرمجيات أكثر تفصيلاً من مناقشة المتطلبات.



لماذا نعتبر الهيكلة كمتطلب أولي؟ لأن كفاءة البناء تحدد مفاهيم السلامة للنظام. وهذا بدوره يحدد الكفاءة النهائية للنظام.

الهيكلة الجيدة تؤمن متطلبات البنية للحفاظ على كفاءة مفاهيم النظام من الأعلى للأسفل. إنها تؤمن التوجيه للمبرمجين– في مستوى من التفاصيل الملائمة لمهارات المبرمجين وللعمل الحالي. وتقسم العمل بحيث تستطيع مجموعة مطورين أو مجموعة فرق تطوير العمل بشكل منفصل. حيث تجعل الهيكلة الجيدة البناء أسهل. بينما تجعل الهيكلة السيئة البناء مستحيل تقريباً.

الشكل 3-7 يوضح مشكلة أخرى مع الهيكلة السيئة.



الشكل 3-7 بدون هيكلة جيدة، فربما يكون لديك المشكلة الصحيحة ولكن الحل الخطأ. ربما يكون من المستحيل الحصول على بناء ناجح.

بيانات مثبتة

تغييرات الهيكلة مكلفة جداً أثناء البناء أو بعده. والوقت المطلوب لإصلاح خطأ في الهيكلة مكافئ للوقت المطلوب لإصلاح خطأ في متطلبات البناء– الأمر هكذا، وهو أكثر من الوقت المطلوب لإصلاح خطأ برمجي (باسيلي وبيريكوني 1984، ويللز 1998)

تغييرات الهيكلة مثل تغييرات المتطلبات والتي تبدو تغييرات صغيرة من الممكن أن تصبح صعبة التحقيق، على أي حال تغييرات الهيكلة تنشأ من الحاجة لإصلاح أخطاء أو لإجراء تحسينات. كلما استطعت تحديد التغييرات بشكل أبكر، كان ذلك أفضل.

مكونات الهيكلة النموذجية[1]:

الكثير من المكونات معروفة لمهندسي النظام الجيدين، إذا كنت تبني كامل النظام بنفسك، فإن عملك في الهيكلة سوف يتداخل مع عملك في التصميم الأكثر تفصيلاً، وفي هذه الحالة، يجب عليك على الأقل التفكير بكل مكون من الهيكلة. إذا كنت تعمل على نظام بنيته معدة من قبل شخص آخر، فيجب عليك أن تكون قادراً على إيجاد المكونات الهامة بدون مساعدة كلب الصيد وقبعة مطاردة الغزلان والمنظار المكبر. في كلا الحالتين هذه هي مكونات الهيكلة للنظر فيها.

تنظيم البرنامج2:

إن أول متطلبات هيكلة النظام هو أخذ نظرة عامة وشاملة لتوصيف النظام بالعموم. وبدون هذه النظرة، ستقضي وقتا صعباً لبناء صورة مترابطة من ألف تفصيل أو حتى من عشرات الصفوف الفردية.

إذا كان النظام لغزاً صغيراً (jigsaw puzzle) مكوناً من 12 قطعة موضوعة بشكل عشوائي فإن طفل في عامه الأول ممكن أن يحله بسهولة. أما لغز من 12 نظام فرعي فربطه مع بعضه سيكون أصعب. وإن لم تستطع ربطه مع بعضه، فلن تستطيع أن تفهم كيف يشارك الصف الذي تطوره بالنظام.

في الهيكلة يجب أن تجد الدليل بأنه انه أخذ بعين الاعتبار بدائل التنظيم النهائي وأن تجد الأسباب لاختيار التنظيم النهائي عوضاً عن بدائله. فإنه من المحبط العمل على صف عندما يبدو كما لو أن قواعد الصف في النظام ليست مفهومة بشكل واضح. من خلال وصف بدائل التنظيم، تؤمن الهيكلة الأساس المنطقي لمنهج التنظيم وتظهر أن كل صف قد تم أخذه بعين الاعتبار بدقة. وجدت مراجعة واحدة لمهارات التصميم بأن الأساس المنطقي للتصميم على الأقل بنفس الأهمية للصيانة كالتصميم بحد ذاته.(رومباش 1990).

[1] **إشارة مرجعية** لمزيد من التفاصيل عن تصميم البرامج ذات المستوى الأدنى، انظر الفصول من 5 حتى 9

[2] إذا لم تكن تستطيع أن تشرح شيئا لطفل عمرة ست سنوات، فأنت بنفسك لا تفهمه حقاً البرت أنشتاين

الهيكلة يجب أن تعرّف الكتلة الرئيسية في البرنامج[1]. معتمدة على حجم البرنامج، كل كتلة بناء يمكن أن تكون صفاً واحداً أو من الممكن أن تكون أنظمة فرعية مكونة من الكثير من الصفوف، وكل كتلة بناء هي صف أو مجموعة من الصفوف أو التكرارات التي تعمل معا في التوابع عالية المستوى كالتفاعل مع المستخدم، وعرض صفحات الوب، ومقاطعة الأوامر، وتغليف قواعد العمل، أو الوصول للبيانات. كل ميّزة موجودة في المتطلبات يجب أن تغطى على الأقل بكتلة بناء واحدة، وإذا تم استدعاء الإجراء من قبل كتلتي بناء أو أكثر، يجب أن تكون هذه الاستدعاءات متكاملة وغير متعاكسة.

يجب أن يكون معرفاً بشكل جيد ما يجب على كل كتلة بناء أن تكون مسؤولة عنه[2]، يجب أن يكون لكتلة البناء منطقة واحدة للمسؤولية، ويجب أن تعرف أقل ما يمكن عن مناطق مسؤولية الكتل الأخرى، وذلك بتقليل ما تعرفه كل كتلة عن الكتل الاخرى، ركز المعلومات حول التصميم إلى كتلة بناء واحدة.

قواعد الاتصال لكل كتلة يجب أن تكون معرّفة بشكل جيد، ويجب أن تحدد الهيكلة أي من الكتل الأخرى يمكن استخدامها بشكل مباشر وأيها بشكل غير مباشر وأيها يجب ألا تستخدم مطلقاً.

الصفوف الرئيسية[3]:

يجب أن تحدد الهيكلة الصفوف الرئيسية التي ستستخدم، ويجب أن تعرّف مسؤولية كل صف رئيسي وكيف سيتفاعل مع الصفوف الأخرى.

ويجب أيضاً أن تتضمن وصفاً لهرمية الصف وحالة التنقلات واستمرارية الأهداف. إذا كان النظام ضخم كفاية فيجب أن يصف كيفية تعريف الصفوف للأنظمة الفرعية.

كما يجب أن تصف الهيكلة تصاميم الصفوف الأخرى التي أخذت بعين الاعتبار وإعطاء أسباب تفضيل التنظيم المختار.

لا تحتاج الهيكلة لتحديد كل صف في النظام،هدف الـ 20/80. قاعدة: تحديد الـ 20% من الصفوف التي تنظم 80% من سلوك النظام (Jacobsen, Booch, and Rumbaugh 1999; Kruchten 2000)

تصميم المعطيات[4]:

[1] **إشارة مرجعية** لمزيد من التفاصيل عن الأحجام المختلفة لكتل البناء في التصميم، انظر "تصميم ليفيلسوف" في المقطع 2.5

2 **إشارة مرجعية** تقليل ما تعرفه كل كتلة بناء عن الكتل الأخرى هو الجزء المفتاحي لإخفاء المعلومات. للمزيد من التفاصيل انظر" إخفاء الأسرار (إخفاء المعلومات)" في المقطع 3.5

[3] **إشارة مرجعية** لمزيد من التفاصيل حول تصميم الصف، انظر الفصل 6 "الصفوف العاملة"

[4] **إشارة مرجعية** لمزيد من التفاصيل حول العمل مع المتغيرات انظر الفصول من 10 حتى 13

يجب أن تصف الهيكلة تصاميم الملفات والجداول الرئيسية ليتم استخدامها. كما يجب عليها توصيف البدائل التي أخذت بالاعتبار وتبرير الخيارات التي تم اتخاذها. فإذا كان التطبيق يحتفظ بسجلات الزبائن واختار المهندس استحضار قائمة من السجلات باستخدام قائمة الوصول التتابعي، يجب أن يشرح المستند لماذا قائمة الوصول التتابعي أفضل من قائمة الوصول العشوائي أو المكّدس أو جدول مختلط. خلال البناء، هذه المعلومات تعطي نظرة ثاقبة لذهن المهندس. خلال الصيانة، نفس هذه النظرة تعتبر مساعدة ثمينة جداً، وبدونها فإنك كمن يشاهد فلم أجنبي بدون ترجمة.

يجب أن يتم الوصول إلى المعطيات بشكل طبيعي من قبل نظام فرعي واحد أو صف، باستثناء وصول الصفوف أو الاجرائيات التكرارية(`routines`) التي تسمح بالوصول للمعطيات بطرق مضبوطة ومجردة. وهذا تم شرحه بالتفصيل في " إخفاء الأسرار (إخفاء المعلومات) " في المقطع `3.5`.

يجب على الهيكلة أن تحدد المستويات العالية للتنظيم ومحتويات أي قاعدة بيانات مستخدمة. ويجب عليها أيضاً شرح لماذا خيار قاعدة معطيات وحيدة مفضل على قواعد بيانات متعددة (أو العكس بالعكس)، وشرح لماذا قاعدة بيانات مفضلة على ملف بسيط (`flat`)، وتحديد التفاعلات الممكنة مع البرامج الأخرى التي تصل لنفس المعطيات، وشرح المناظير (`views`) [1]التي وضعت على المعطيات، وهكذا.

قواعد العمل:

إذا اعتمدت الهيكلة على قواعد عمل محددة فيجب تحديد هذه القواعد ووصف أثرها على تصميم النظام، مثلاً افترض أن النظام يتطلب اتباع قواعد عمل بأن معلومات الزبون يجب أن لا تبقى بحاجة تحديث أكثر من 30 ثانية، في هذه الحالة يجب وصف تأثير هذه القاعدة على نهج الهيكلة في الحفاظ على معلومات الزبون محدثة ومتزامنة.

تصميم واجهة المستخدم:

تصنف واجهة المستخدم غالباً على أنها من الاحتياجات، إن لم تكن كذلك، فيجب أن تصنف في هيكلة البرمجيات. يجب على الهيكلة أن تحدد العناصر الرئيسية لصيغ صفحات الانترنت، والواجهات الرسومية وواجهة سطر الأوامر وغيرها.

1 **المناظير**: يعتبر المنظار نافذة نتمكن من خلالها من الوصول إلى أعمدة وأسطر محددة من الجدول ويستخدم لإعطاء المستخدمين صلاحيات للوصول إلى هذه الأعمدة والأسطر دون غيرها والمنظار لا يحوي أي بيانات إنما هو عبارة عن طريقة للوصول إلى معطيات محددة من الجدول نفسه. كما يستخدم المنظار لإجراء الاستفسارات المعقدة حيث يمكن تقسيم الأعمدة الموجودة في الجدول والمترابطة فيما بينها إلى مناظير مستقلة تساعد في تبسيط الاستفسارات.

الهيكلة الدقيقة لواجهة المستخدم تصنع الفرق بين البرامج المحببة والبرامج التي لا تستخدم أبداً. الهيكلة يجب أن تكون مجزأة بحيث يمكن استبدال واجهة مستخدم جديدة بدون التأثير على قواعد العمل وعلى أجزاء الخرج للبرنامج.

مثلاً: يجب على الهيكلة أن تجعل من السهل تجميع مجموعة من صفوف واجهات المستخدم التفاعلية والدخول لمجموعة صفوف سطر الأوامر. هذه الإمكانية غالباً ما تكون مفيدة خصوصاً أن واجهات سطر الأوامر ملائمة لفحص البرمجيات في الوحدة أو مستوى النظام الفرعي.

إن تصميم واجهات المستخدم يستحق أن يكون له كتاب طويل لمناقشته[1]، ولكن هذا خارج تركيز الكتاب.

إدارة الموارد:

يجب على الهيكلة أن تضع خطة لإدارة الموارد النادرة كاتصالات قاعدة البيانات، والتهديدات، والتعاملات. إدارة الذاكرة هي مجال آخر مهم للهيكلة للتعامل مع مجالات تطبيقات الذاكرة المقيدة كتطوير التعاريف والأنظمة المدمجة. ويجب على الهيكلة أن تثمن الموارد في الحالات الصغيرة والكبيرة. في حالة بسيطة، يجب على التقييم أن يظهر بأن المصادر المطلوبة جيدة من خلال إمكانية تنفيذ البيئة المطلوبة. وفي حالة معقدة أكثر ربما يحتاج التطبيق لإدارة موارده الخاصة بفعالية، إن كان كذلك، فإن إدارة الموارد يجب أن تكون مبنيّة بعناية كأي جزء من النظام.

الأمن[2]:

يجب على الهيكلة أن تصف النهج المتبع للأمن على مستوى التصميم وعلى مستوى الشفرة. إذا كان نموذج التهديد لم يبنى بعد، فإنه يجب أن يبنى في وقت البرمجة، مبادئ كتابة الشفرة يجب تطويرها في عقلنا مع مراعاة تضمينها القواعد الأمنية، بما في ذلك نهج التعامل مع الذاكرة المؤقتة وقواعد التعامل مع البيانات غير الموثوقة (بيانات مدخلة عن طريق المستخدمين وملفات تعريف الارتباط "الكعكات" (`cookies`) وبيانات التعريف ومختلف الواجهات الخارجية الأخرى) والتشفير ومستوى التفاصيل المضمنة في رسائل الخطأ وحماية البيانات السرية في الذاكرة والعديد من القضايا الأخرى.

[1] cc2e.com/0393

[2] cc2e.com/0330

لمزيد من القراءة لمناقشة ممتازة لأمن البرمجيات، انظر كتابة شفرة آمنة، الإصدار الثاني. (Howard and LeBlanc 2003) وكذلك عدد كانون الثاني 2002 باب برمجيات IEEE

الأداء[1]:

إذا كان الأداء مصدر قلق، فيجب أن تحدد أهداف الأداء في المتطلبات. أهداف الأداء يمكن أن تشمل استخدام الموارد، وفي هذه الحالة الأهداف يجب أن تحدد أيضاً الأولوية بين المصادر، بما في ذلك السرعة مقابل الذاكرة مقابل التكلفة.

ويجب على الهيكلة أن توفر التقديرات وتشرح لماذا يؤمن المهندسون بأن الأهداف قابلة للتطبيق. إذا كانت مناطق محددة مهددة بخطر الفشل بتحقيق أهدافها، فيجب على الهيكلة أن تقول ذلك، وإذا كانت مناطق محددة تتطلب استخدام خوارزميات أو أنواع المعطيات لتحقيق أهداف الأداء، فيجب على الهيكلة أن تقول ذلك أيضاً، تستطيع الهيكلة أيضاً تضمين ميزانيات المكان والزمان لكل صف أو كائن.

قابلية التوسع:

قابلية التوسع هي قدرة النظام على النمو لتحقيق مطالب مستقبلية، ويجب أن تصف الهيكلة كيف سيعالج النظام النمو في عدد المستخدمين، وعدد الخدمات، وعدد خوادم الشبكة، وعدد سجلات قاعدة البيانات، وحجم سجلات قاعدة البيانات، وحجم المناقلات (transaction)، وهكذا. إذا كان من غير المتوقع نمو النظام وتوسعة فهي ليست مشكلة، ويجب على الهيكلة أن تصرّح عن هذا الافتراض.

المشاركة بالعمل:

إذا كان من المتوقع أن يشارك النظام البيانات أو الموارد مع البرامج أو الأجهزة الأخرى، ينبغي أن تصف الهيكلة كيفية إنجاز ذلك.

التدويل/ المحلية:

"التدويل" هو الفعالية التقنية لإعداد برنامج يدعم محليات متعددة.

العالمي او التدويل يعرف عادة بـ /I18N/ لان الحرف الأول والأخير في كلمة "Internationalization" هما "I" و"N" وبسبب وجود 18 حرفاً بينهما في وسط الكلمة، المحلي "Localization" يعرف عادة بـ /L10N/ لنفس السبب، وهو إمكانية ترجمة البرنامج لدعم لغة محلية محددة. النسخ الدولية تستحق الاهتمام في الهيكلة من أجل نظام متفاعل.

معظم الأنظمة التفاعلية تتضمن عشرات أو مئات الدالات وعروض الحالات، ورسائل المساعدة ورسائل الخطأ، وهكذا، الموارد المستخدمة في السلاسل المحرفية يجب أن تكون مقدرة سلفاً، إذا كان البرنامج سيستخدم

[1] **لمزيد من القراءة** لمعلومات إضافية لتصميم الأنظمة من أجل الأداء، انظر هندسة الأداء لأنظمة البرمجيات لكوني سميث (1990).

تجارياً فيجب على الهيكلة أن تبين بأن السلاسل القياسية ونسخ مجموعة المحارف قد أخذت بعين الاعتبار، متضمنة مجموعة المحارف المستخدمة (ASCII، DBCS، EBCDIC، MBCS، Unicode، ISO8859 وهكذا)، وأنواع من السلاسل المحرفية المستخدمة (محارف لغة الـ C، محارف لغة فيجول بيزك، وغيرها)، والمحافظة على السلاسل المحرفية بدون تغيير الشفرة، وترجمة السلاسل المحرفية إلى لغة أجنبية بأقل تأثير على الشفرة وواجهة المستخدم.

تستطيع الهيكلة أن تقرر استخدام سلسلة المحارف بسطر في الشفرة عند الحاجة، أو ابقاء سلسلة المحارف في صف والإشارة إليها من خلال واجهة الصف، أو تخزين السلسلة في ملف مصدر. يجب أن تشرح الهيكلة أي خيار تم اختياره ولماذا.

المدخلات/ المخرجات

المدخلات / المخرجات (I/O) هي مجال آخر يستحق الاهتمام في الهيكلة، يجب على الهيكلة أن تخصص نظرة للأمام ونظرة للخلف عند قراءة المخطط. ويجب عليها وصف المستوى الذي تم اكتشاف أخطاء الإدخال والإخراج فيه، في الحقل أو السجل أو الجدول، أو في مستوى الملف.

معالجة الأخطاء:

بيانات مثبتة

تتجه معالجة الأخطاء لأن تكون واحدة من المشاكل الشائكة لعلوم الحاسوب الحديثة، ولا يمكن التعامل معها عشوائياً. يقدر بعض الناس بأن أكثر من 90% من شفرة البرامج كتبت للاستثناءات وحالات معالجة الأخطاء أو الحفاظ على الهيكلة، مما يتضمن بأن 10% من شفرة البرامج كتبت لحالات اعتبارية. (Shaw in Bentley 1982). مع الكثير من الشفرة المخصصة للتعامل مع الأخطاء، في الهيكلة يجب وضع استراتيجية التعامل معهم باستمرار.

على أي حال، إذا تمت معالجة الاخطاء فغالباً ما يتم التعامل معها كقضية على مستوى اتفاقية كتابة الشفرة، ولكن لأن لها آثار على نطاق النظام، فمن الأفضل معاملتها على مستوى الهيكلة.

هنا بعض الأسئلة التي يجب أخذها بعين الاعتبار:

- هل معالجة الأخطاء للإصلاح أم أنه للكشف فقط؟ إذا كان للإصلاح، يمكن للبرنامج محاولة الاستعادة من الأخطاء. إذا كان للكشف فقط، يستطيع البرنامج متابعة المعالجة كأن شيئا لم يكن، في هذه الحالة يجب عليه إعلام المستخدم بأن خطا ما قد اكتشف.
- هل اكتشاف الأخطاء فعّال ام غير فعّال؟ يستطيع النظام التنبؤ بحدوث الأخطاء- مثلاً: فحص صحة مدخلات المستخدم- أو يستجيب لهم بشكل سلبي عندما لا يستطيع تجنبهم- مثلاً: عند جمع مدخلات

المســتخدم ينتج فائض في التدفق الرقمي. اكتشــاف الأخطاء يمكن أن يخلي لك الطريق أو ينظف الفوضى. مرة أخرى، في كلا الحالتين يتضمن الخيار واجهة المستخدم.

- كيف ينقل البرنامج الأخطاء؟ عند اكتشــاف خطأ، يســتطيع وعلى الفور أن يوصــف المعطيات التي ســببت الخطأ، يســتطيع التعامل مع الخطأ كخطأ والدخول في وضــعية تصــحيح الأخطاء، أو يمكنه الانتظار حتى انتهاء العملية وإعلام المستخدم بأنه تم اكتشاف خطأ (بطريقة ما).
- ما هي آلية التعامل مع رسائل الخطأ؟ إذا لم تحدد الهيكلة استراتيجية محددة ومتماسكة، فإن واجهة المســتخدم ســتبدو مرتبكة كالخلط بين المعكرونة والفول المجفف من الواجهات المختلفة في أجزاء مختلفة من البرنامج، لتجنب هذا المظهر، على الهيكلة تأسيس اصطلاحات لرسائل الخطأ.
- كيف يتم التعامل مع الاســتثناءات؟ على الهيكلة تحديد متى ســتعبر الشــفرة الاســتثناءات، وفيما إذا كان سيتم توقيفهم، وكيف سيدخل عليهم، وكيف سيتم توثيق ذلك، وهكذا.
- داخل البرنامج، بأي مســتوى ســيتم التعامل مع الأخطاء[1]؟ يمكن التعامل معهم عند اكتشــافهم، أو تحويلهم إلى صف التعامل مع الأخطاء، أو تحويلهم إلى قائمة الاستدعاء.
- ماهي مسـؤولية كل صـف لجهة شـرعية مدخلاتها؟ هل كل صـف مسـؤول عن إثبات معطياته، أم هل مجموعة من الصفوف مسؤولة عن إثبات معطيات النظام؟ هل تستطيع الصفوف بأي مستوى تأكيد أن المعطيات التي تتلقاها نظيفة؟
- هل تريد اســتخدام آلية البيئة المدمجة للتعامل مع الأخطاء، أو بناء البيئة الخاصــة بك؟ الحقيقة بأن بيئة لديها النهج الخاص بها للتعامل مع الأخطاء لا يعني بأنه أفضل نهج لمتطلباتك.

التسامح مع الخطأ[2]:

يجب على الهيكلة أيضاً الإشارة لنوع التسامح مع الخطأ المتوقع. نسبة الخطأ المسموحة هي مجموعة من التقنيات التي تزيد من فاعلية النظام بالكشف عن الأخطاء، والاسترداد منها إن أمكن، واحتواء آثارها السلبية.

مثلاً: يستطيع النظام حساب الجذر التربيعي لعدد مرات التسامح بالخطأ بإحدى هذه الطرق:

- ربما يقوم النظام بعمل نســخة احتياطية والمحاولة ثانية عند اكتشــاف خطأ، إذا كان الجواب الأول خطأ، فأنه سيعود للنقطة التي يعرف فيها بأن كل شيء كان وقتها صحيح، ويتابع منها.

[1] **إشارة مرجعية** منهج متناسق للتعامل مع المحددات السيئة هو جانب أخر من استراتيجية معالجة الأخطاء التي ينبغي أن تعالج بطريقة هيكلية. على سبيل المثال، انظر الفصل 8 "البرمجة الدفاعية"

[2] **لمزيد من القراءة** للحصول على مقدمة جيدة لتسامح الخطأ، انظر إلى عدد تموز 2001 لبرمجيات IEEE. بالإضافة إلى توفير مقدمة جيدة، تستشهد المقالات بالعديد من الكتب والمقالات المفتاحية حول هذا الموضوع.

- يمكن للنظام أن يملك شــفرة برمجية إضــافية لاســتخدامها عن اكتشــاف خطأ في الترميز البرمجي الأساسي. في المثال، إذا كان الجواب الأول خاطئ فإن النظام سيبدل إلى جذر تربيعي ويستخدمه.
- ربما يســتخدم النظام خوارزمية التصــويت. والتي من الممكن أن يكون لها صــفوف من ثلاث جذور تربيعية والتي يســتخدم كل منها طريقة مختلفة. كل صــف يحســب الجذر التربيعي، وبعدها يقوم النظام بمقارنة النتائج. معتمداً على نســبة الخطأ المســموحة المضــمّنة في النظام. وعندها يســتخدم الوسط أو المتوسط أو أسلوب النتائج الثلاثة.
- قد يســتبدل النظام القيمة الخاطئة بالقيمة المزيفة التي يعرفها للحصــول على آثار مفيدة على بقية النظام.

للتسامح مع الخطأ فروع أخرى تتضمن حمل النظام ليتغير إلى حالة التشغيل الجزئي او الحالة الوظيفية المتدهورة عند اكتشاف خطأ. بإمكانه ايقاف نفسه أو أعادة التشغيل أوتوماتيكياً. هذه الأمثلة بالضرورة مبسطة. التسامح مع الخطأ هو موضع رائع ومعقد، وهو لسوء الحظ خارج نطاق هذا الكتاب.

جدوى الهيكلة:

قد يقلق المصمم حول قابلية النظام لمواجهة كفاءة أهدافه، والعمل ضمن المصادر المحدودة، أو أن يكون مدعوماً بشكل ملائم من بيئة التنفيذ. يجب على الهيكلة إثبات أن النظام ملائم تقنياً. إذا كانت عدم الملائمة في أي منطقة قادرة على جعل المشروع غير قابل للعمل، فيجب أن تشير الهيكلة لكيفية التحقيق في هذه المسائل من خلال النماذج الأولية أو البحث أو وسائل أخرى. يجب حل هذه المخاطر قبل بدء بالهيكلة الكاملة.

الافراط في الهندسة:

المتانة هي قدرة النظام على الاستمرار في العمل بعد أن يكتشف خطأ. غالباً تحدد الهيكلة نظاماً أكثر متانةً من النظام الذي تحدده المتطلبات. أحد الأسباب هو أن نظام مؤلف من أجزاء كثيرة بمتانة دنيا من الممكن أن يكون عموماً أقل قوة من المطلوب.

في البرمجيات، السلسلة ليست بقوة أضعف حلقة، بل بضعف كل الحلقات الضعيفة مجتمعة. يجب أن تشير الهيكلة بوضوح ما إذا كان المبرمجون سيخطئون من جهة الهندسة الفائقة أو من جهة فعل أبسط الأشياء التي تعمل.

تحديد نهج للهندسة الفائقة مهم بشكل واضح لأن الكثير من المبرمجين يهندسون صفوفهم بشكل مفرط أوتوماتيكيا بدون الشعور بالاعتزاز. عبر وضع توقعات واضحة في الهيكلة، يمكنك تجنب ظاهرة أن بعض الصفوف متينة بشكل جيد والبعض الآخر بالكاد متانته كافيه.

الشراء مقابل بناء القرارات[1]

الحل الأكثر تطرفاً لبناء البرمجيات هو عدم بناءها على الإطلاق بل بشرائها أو تحميل برمجيات مفتوحة المصدر مجاناً. يمكنك شراء ضوابط واجهة المستخدم الرسومية(GUI) ومدراء قواعد البيانات ومعالجات الصورة والرسوميات ومكونات الجداول والأمن وتشفير المكونات وأدوات الجدول وأدوات معالجة النصوص، القائمة لا نهائية تقريباً. أحد أهم فوائد البرمجة في بيئة واجهة المستخدم الرسومية الحديثة هي كمية التوظيف التي تحصل عليها أوتوماتيكياً: صفوف الرسوميات ومدراء صندوق الحوار ولوحة المفاتيح وبدائل للفأرة (mouse)، والشفرات البرمجية التي تعمل أوتوماتيكياً مع أي طابعة أو شاشة وهكذا.

إذا كانت الهيكلة لا تستخدم مكونات جاهزة فيجب شرح الطرق التي فيها اعتماد على مكوّنات البناء المخصصة لتجاوز المكتبات والمكوّنات الجاهزة.

إعادة استخدام القرارات:

إذا كانت الخطة تدعو إلى استخدام البرامج الموجودة مسبقاً أو حالات الاختبار أو صيغ البيانات أو غيرها من المواد، فإن على الهيكلة شرح كيف للبرمجيات المعاد استخدامها أن تتطابق مع أهداف الهيكلة الأخرى، إذا كانت ستنفذ لأجل المطابقة.

استراتيجية التغيير:

لأن بناء المنتجات البرمجية هو عمليّة تعليميّة لكل من المبرمجين والمستخدمين، فالمنتج يتغيّر خلال فترة تطويره[2]. تظهر التغييرات من أنواع البيانات المختلفة وصيغ الملفات والوظائف المتغيرة والميزات الجديدة وغيرها.

قد تكون التغيرات قدرات جديدة محتملة نتيجة تحسينات التخطيط، أو يمكن أن تكون قدرات لم تقم بعملها في النسخة الأولى من النظام. وهكذا فان أحد أهم التحديات التي تواجه مهندس البرمجيات هي جعل الهيكلة ثابتة بما يكفي للتكيّف مع التغييرات المحتملة.

---

[1] **إشارة مرجعية** للحصول على قائمة بأنواع المكونات والمكتبات المتاحة تجارياً، انظر "مكتبات الشفرة" في المقطع 3.30

[2] **إشارة مرجعية** لمزيد من التفاصيل جول التعامل مع التغيرات بشكل منظم، انظر المقطع 2.28 "إدارة الضبط"

يجب أن تصف الهيكلة بوضوح استراتيجية التعامل مع التغييرات[1]. ويجب أن تبيّن أن التحسينات الممكنة قد أُخذت بعين الاعتبار، وأن التحسينات الأكثر احتمالاً هي أيضاً الأسهل للتنفيذ. إذا كانت التغييرات محتملة في صيغ الدخل أو الخرج أو أسلوب تفاعل المستخدم أو متطلبات المعالجة فيجب على الهيكلة أن تبيّن بأن كل التغييرات متوقعة وأن آثار أي تغيير ستكون محدودة بعدد قليل من الصفوف، خطة الهيكلة للتغييرات يمكن أن تكون من السهولة بالشخص لوضع أرقام الإصدار في ملفات البيانات، تحفظ الحقول للاستخدام المستقبلي، أو تصميم ملفات بحيث تستطيع إضافة جداول جديدة، إذا أُستخدم مولد الشفرة، فيجب على الهيكلة إظهار أن التغييرات المتوقعة هي ضمن إمكانية مولد الشفرة.

يجب على الهيكلة الإشارة للاستراتيجية المستخدمة لتأخير الالتزام[2]. مثلاً: ربما تحدد الهيكلة بأن العمل على تقنية مقادة بالجداول يجب أن تستخدم أكثر من اختبارات `if` الدقيقة. وقد تحدد بأن معطيات هذه الجداول يجب أن تحفظ في ملفات خارجية أكثر من أن تشفير داخل البرنامج، وهكذا تسمح للتغييرات في البرنامج بدون إعادة الترجمة (`recompiling`).

الجودة العامة للهيكلة[3]:

تمتاز مواصفات الهيكلة الجيدة بدراسة الصفوف في النظام للمعلومات المخبأة في كل صف، والأساس المنطقي لإدراج واستبعاد كل بدائل التصميم الممكنة.

وينبغي أن تكون الهيكلة وحدة مفاهيم مصقولة مع بعض الإضافات المخصصة. الفرضية المركزية لأكثر كتب هندسة البرمجيات شعبية حتى الآن / `The Mythical Man-Month` / بأن المشكلة الأساسية مع الأنظمة الكبيرة هي الحفاظ على سلامة مفاهيمها (`Brooks 1995`). الهيكلة الجيدة يجب أن تكون مناسبة للمشكلة. فعندما تلقي نظرة على الهيكلة يجب أن تكون مسروراً لأن الحل يبدو طبيعياً وسهلاً، ولا يجب أن يبدو أن الهيكلة والمشكلة غير متناسبان وكأنه قد تم ربطهما معاً بشريط لاصق.

ربما تعرف الطرق التي تغيرت بها الهيكلة خلال تطويرها، فكل تغيير يجب أن يتناسب تماماً مع الفكرة العامة. يجب ألا تبدو الهيكلة كالاعتمادات المالية للكونغرس الامريكي تتم بالهدايا، ويوجد وسطاء للتوظيف في منطقة سكن كل نائب.

[1] أخطاء التصميم غالبا ما تكون غير متوقعة وتحدث خلال التقدم مع نسيان الافتراضات السابقة عند إضافة ميزات أو استخدامات جديدة للنظام فرناندو ج. كوربات

[2] **إشارة مرجعية** للحصول على شرح كامل عن تأخير الالتزام، انظر "اختر التقيد بالوقت بوعي" في المقطع 3.5

[3] **إشارة مرجعية** لمزيد من المعلومات حول كيفية تفاعل سمات الجودة، انظر المقطع 1.20،"معالم جودة البرمجيات"

يجب أن تكون عناصر الهيكلة محددة بوضوح. فالتصميم لنظام مع هدف رئيسي لقابلية التعديل سيكون مختلفاً عن هدف الأداء غير القابل للحل، حتى لوكان كلا النظامين يملكان نفس التابع(function). يجب على الهيكلة وصف التحريض لكل القرارات الحاسمة. كن حذراً من أعذار "نحن دائما نفعل ذلك بهذه الطريقة". إحدى القصص تقول بأن بيث أرادت أن تطبخ السمك بالمقلاة بناء على وصفة ٍ مشهورة لتقدمها لأهل زوجها، اتصلت بصديقتها لأخذ الوصفة فأخبرتها أن أمها قالت بأنه يتوجب رش الملح والفلفل عليها وتقطيع نهايتيها ووضعها في المقلاة وتغطيتها وطهيها. سألتها بيث: لماذا يجب تقطيع نهايتيها؟، أجابت " لا أعلم، نحن دائما نفعل ذلك بهذه الطريقة، دعيني اسأل أمي " اتصلت بها، فقالت لها " لا أعلم أنا دائماً أقوم بها بهذه الطريقة، دعني أسال جدتك"، اتصلت بجدتها، والتي أجابتها بكل بساطة وهدوء: "كانت حياتنا بسيطة وقدراتنا متواضعه ولم يكن لدي سوى مقلاه صغيره لا تتسع لسمكه كامله !!"

بصورة عامة الهيكلة البرمجية الجيدة هي آلة – ولغتها- مستقلة باعتراف الجميع. فيجب عليك أن تكون مستقلاً عن البيئة حتى تقام إغراء زيادة هيكلة النظام، أو أن تقوم بالعمل الذي يمكنك القيام به بشكل أفضل خلال البناء. إذا كانت غاية البرنامج فحص آلة محددة أو لغة محددة، فإن هذه النصائح لا تعمل.

يجب أن تسير الهيكلة على الخط الواقع بين قلّة التوصيف والتوصيف الزائد للنظام. ولا يجب لأي جزء من الهيكلة أن يحظى باهتمامٍ أكثر مما يستحق، أو أن يُهتم بتصميمه بشكل زائد. كما لا يجب على المصممين توجيه الاهتمام لجزء دون الآخر. على الهيكلة أن تعنون كل المتطلبات بدون صفيحة ذهبية (بدون أن تحتوي العناصر غير المطلوبة).

يجب على الهيكلة أن تحدد بوضوح مكامن الخطر. ويجب عليها أيضاً شرح لماذا هي خطيرة، وما هي الخطوات المتخذة لتقليل خطرها. كما يجب أن تتضمن العديد من الرؤى. خطط بناء منزل ستتضمن مساقط رأسية وخطة للأرضية وخطة للزراعة ومخططات بيانية للكهرباء وكل الرؤى الأخرى لبناء المنزل. وكذلك الهيكلة البرمجية تصف فوائد تزويد النظام بالرؤى المختلفة للنظام التي تبعد الأخطاء والتعارضات وتساعد المبرمجين على فهم تصميم النظام بشكل كامل. (Kruchten 1995).

أخيراً: يجب ألا تكون قلقاً حول أي جزء من الهيكلة. فيجب ألا تتضمن الهيكلة أي شيء وُضِع فقط لإرضاء المدير. كما يجب ألا تتضمن أي شيء يصعب عليك فهمه. فأنت الشخص الذي سينفذها، إذا لم يعني لك هذا شيء، كيف لك ان تنفذها؟

قائمة تدقيق: الهيكلة

فيما يلي قائمة بالقضايا التي يجب أن تعالجها الهيكلة الجيدة. القائمة لا تهدف إلى أن تكون دليلاً شاملاً للهيكلة ولكن لأن تكون وسيلة واقعية من تقييم المحتوى الغذائي لما تحصل عليه في نهاية برمجة السلسلة الغذائية البرمجيات. استخدم قائمة التحقق هذه كنقطة انطلاق لقائمة تدقيق خاصة بك.

كما هو الحال مع قائمة المتطلبات، إذا كنت تعمل على مشروع غير رسمي، ستجد بعض العناصر التي لا تحتاج حتى للتفكير. إذا كنت تعمل على مشروع أكبر، فإن معظم البنود ستكون مفيدة.

موضوعات هيكلة محددة

- هل التنظيم الشامل للبرنامج واضح، بما في ذلك النظرة العامة ومبررات هيكلة جيدة؟
- هل تم تحديد كتل البناء الرئيسـية بشـكل جيد، بما في ذلك مجالات مسـؤوليتها وواجهاتها مع كتل البناء الأخرى؟
- هل جميع الوظائف المدرجة في المتطلبات مغطاة بشكل معقول، من قبل عدد كبير جداً أو عدد قليل جداً من الكتل؟
- هل الفئات الأكثر أهمية موضحة ومبررة؟
- هل تم وصف تصميم البيانات وتبريره؟
- هل تم تحديد تنظيم قاعدة البيانات والمحتوى؟
- هل تم تحديد جميع قواعد العمل الرئيسية ووصف أثرها على النظام؟
- هل تصف استراتيجية تصميم واجهة المستخدم؟
- هل واجهة المستخدم منظمّة بحيث أنّ التغييرات فيها لن تؤثر على بقية البرنامج؟
- هل تصف استراتيجية التعامل مع I/O ومبرراتها؟
- هل تقديرات استخدام الموارد واستراتيجية إدارة الموارد وصفت وتم تبريرها للموارد الشحيحة مثل المواضيع، ووصلات قاعدة البيانات، والمقابض، وعرض النطاق الترددي للشبكة، وهلم جرا؟
- هل تم وصف متطلبات الأمن الخاصة بالهيكلة؟
- هل تحدد الهيكلة ميزانيات المساحة والسرعة لكل فئة أو نظام فرعي أو منطقة وظيفية؟
- هل تصف الهيكلة كيفية تحقيق قابلية التوسع؟
- هل تعالج الهيكلة التوافقية؟
- هل وصفت استراتيجية التدويل / المحلي؟
- هل توفر استراتيجية متماسكة لمعالجة الأخطاء؟
- هل تم تحديد نهج التسامح مع الخطأ (إذا كان هناك حاجة إلى ذلك)؟
- هل تم وضع جدوى تقنية لجميع أجزاء النظام؟
- هل تم تحديد نهج في الهندسة الفائقة؟
- هل من الضروري اتخاذ قرارات الشراء مقابل البناء؟

- هل تصف الهيكلة الكيفية التي سيتم بها استخدام الشفرة المعاد استخدامها لأهداف هيكلة أخرى؟
- هل صممت الهيكلة للاستيعاب التغيرات المحتملة؟

كفاءة الهيكلة العامة

- هل تمثل الهيكلة جميع المتطلبات؟
- هل أي جزء فوق الهيكلة أو تحت الهيكلة؟ هل وضعت هذه التوقعات خارج الاعتبار بشكل صريح؟
- هل الهيكلة كلها متوافقة مع بعضها من الناحية النظرية؟
- هل التصميم على مستوى عال مستقل عن الجهاز واللغة التي سوف يتم استخدامهما لتنفيذه؟
- هل الدوافع لجميع القرارات الرئيسية مقدّمة؟
- هل أنت، كمبرمج ستقوم بتنفيذ النظام، مرتاح للهيكلة؟

6.3 الوقت المطلوب لإنجاز للمتطلبات الأولية التحضيرية[1]

يختلف مقدار الوقت الذي تقضيه على تعريف المشكلة، والمتطلبات، وهندسة البرمجيات وفقا لاحتياجات المشروع الخاص بك. وبشكل عام، يكرّس مشروع جيد المدى ما يتراوح بين 10 و20 في المائة من جهوده وحوالي 20 إلى 30 في المائة من جدوله الزمني للمتطلبات والهيكلة والتخطيط المسبق (ماكونيل 1998، كروشتن 2000). وهذه الأرقام لا تشمل وقت التصميم التفصيلي – والذي يعتبر جزءاً من البناء.

إذا كانت المتطلبات غير مستقرة وكنت تعمل على مشروع رسمي كبير، ربما يجب عليك أن تعمل مع محلل المتطلبات لحل المشاكل التي يتم تحديدها في وقت مبكر من البناء. مما يتيح الوقت لك للتشاور مع محلل المتطلبات، ولمحلل المتطلبات لمراجعة المتطلبات قبل أن يصبح لديك نسخة قابلة للتطبيق من المتطلبات.

وإذا كانت المتطلبات غير مستقرة وكنت تعمل على مشروع صغير غير رسمي، فربما تحتاج إلى حل مشكلات المتطلبات بنفسك. وإتاحة الوقت اللازم لتحديد المتطلبات بشكل جيد بحيث يكون لتقلبها تأثير ضئيل على البناء.

[1] **إشارة مرجعية** مقدار الوقت الذي تقضيه على المتطلبات الأساسية يعتمد على نوع المشروع الخاص بك. لمزيد من التفاصيل عن تكييف المتطلبات الأساسية الخاصة بك انظر القسم 3.2، "تحديد نوع البرمجيات التي تعمل عليها" في وقت سابق من هذا الفصل.

في حال كانت المتطلبات غير مستقرة -سواء أكنت تعمل على مشروع رسمي أو غير رسمي- عالج عمل المتطلبات كمشروع خاص[1]. وقدّر الوقت المتوقع لبقية المشروع بعد انتهاءك من المتطلبات.

منطقياً خلال هذه المقاربة الحساسة لا أحد يمكنه توقع جدولك الزمني للعمل قبل أن تعرف أنت "ما الذي تريد فعله"؟ إنها كما لو كنت متعهد بناء دُعيت للعمل على منزل. يقول لك الزبون "ما هي تكلفة هذا العمل"؟ تجيبه منطقياً بسؤال، "ما الذي تريدني أن أفعله؟" يقول الزبون، "لا أستطيع إخبارك، ولكن كم ستكلفني؟" منطقياً تشكر الزبون على إضاعته لوقتك وتعود للمنزل.

بالنسبة للبناء، إنّه من غير المنطقي أن يسألك الزبون عن التكلفة قبل أن يخبرك بما يتوجب عليك بناءه. لا يريد زبائنك منك أن تَحضُر ومعك الاخشاب والمطرقة والمسامير وأن يبدؤا بصرف نقودهم قبل أن ينهي المعماري مخطط المشروع. توجه الناس لفهم تطوير البرمجيات أقل من فهمهم له. لذلك فإن زبائنك ليس من الضرورة أن يتفهموا رغبتك في التخطيط لتطوير المتطلبات كمشروع منفصل.

من الممكن أن تكون بحاجة لشرح أسبابك لهم. عندما تخصص وقتاً لهيكلة البرمجيات، استخدم نهج شبيه بذلك المستخدم في تطوير المتطلبات، في حال كانت البرمجية من النوع الذي لم تتعامل معه مسبقاً، اسمح بهامش وقت إضافي للتأكد من التصميم في مكان ما. تأكد من أن الوقت الذي تحتاجه لإنشاء هيكلة جيدة لن يأخذ كثيراً من الوقت الذي ستحتاجه للقيام بالعمل جيداً في مناطق أخرى. إذا لزم الأمر، خطط للهيكلة كمشروع منفصل أيضاً.

مصادر إضافية

فيما يلي المزيد من المصادر حول المتطلبات[2]:

المتطلبات[3]

يوجد هنا عدة كتب تحوي تفاصيل أكثر عن تطوير المتطلبات:

[1] **إشارة مرجعية**: للتعرف على طرق التعامل مع المتطلبات المتغيرة، راجع "تغييرات المتطلبات أثناء البناء" في القسم 3.4، في وقت سابق في هذا الفصل.

[2] cc2e.com/0344

[3] cc2e.com/0351

Wiegers ،Karl. (متطلبات البرمجيات)Software Requirements2 ،d ed. Redmond ،WA: Microsoft Press2003 ،.

إنه كتاب عملي، تركيز الكتاب المهني على "براغي وصواميل" أنشطة المتطلبات، متضمناً استنباط المتطلبات، وتحليل المتطلبات، وتحديد المتطلبات، والتحقق من المتطلبات، وإدارة المتطلبات.

Robertson ،Suzanne and James Robertson. (اتقان عملية المتطلبات)Mastering the Requirements Process. Reading ،MA: Addison-Wesley1999 ،.

يعتبر بديل جيد لكتاب Wiegers للممارس المتقدم للمتطلبات.

Gilb ،Tom. (الهندسة التنافسية)Competitive Engineering. Reading ،MA: Addison-Wesley، 2004.

يصف هذا الكتاب متطلبات لغة Gilb ويعرف بـ "Planguage." يغطي هذا الكتاب نهج محددة لهندسة المتطلبات، والتصميم وتقييم التصميم، وإدارة المشروع التطورّية. تستطيع تحميل الكتاب من الموقع الإلكتروني من خلال الرابط www.gilb.com.[1]

IEEE Std 830-1998. IEEE المتطلبات الموصى بها لمواصفات متطلبات البرمجيات.

Los Alamitos ،CA: IEEE Computer Society Press هذه الوثيقة هي دليل الـ IEEE-ANSI لكتابة مواصفات متطلبات البرمجيات. فهي تصف ما لذي يتوجب تضمينه في وثيقة المواصفات وتضع عدة الخطوط العريضة البديلة.

Abran ،Alain ،Swebok. et al. دليل معرفة هندسة البرمجيات.

Los Alamitos ،CA: IEEE Computer Society Press ،2001. ويحتوي على وصف لتفاصيل معرفة هندسة البرمجيات. يمكن تحميله ايضاً من www.swebok.org.[2]

فيما يلي بدائل أخرى جيدة:

Lauesen ،Soren. Software Requirements(متطلبات البرمجيات): Styles and Techniques(النمط والتقنية). Boston ،MA: Addison

Wesley2002 .

Kovitz ،Benjamin L. Practical Software Requirements(متطلبات البرمجيات العملية): A Manual of Content and Style(كراسة المحتوى والنمط).

Manning Publications Company1998 .

[1] cc2e.com/0358

[2] cc2e.com/0365

Cockburn ،Alistair. Writing Effective Use Cases(كتابة حالات الاستخدام الفعّالة). Boston ،MA: Addison-Wesley2000 .

هيكلية البرمجيات [1]

تم نشر العديد من الكتب عن هيكلة البرمجيات في السنوات القليلة الماضية. وهنا بعض من أفضلها:

Bass ،Len ،Paul Clements ،and Rick Kazman. (هيكلة البرمجيات في المهنة)Software Architecture in Practice2 ،d ed.

Boston ،MA: Addison-Wesley2003 ،.

Buschman ،Frank ،et al. (هيكلة البرمجيات نمطية التوجّه)Pattern-Oriented Software Architecture ،Volume 1:

A System of Patterns. New York ،NY: John Wiley & Sons1996 ،.

Clements ،Paul ،ed. (توثيق هيكلة البرمجيات)Documenting Software Architectures: Views and Beyond. Boston ،MA:

Addison-Wesley2003 ،.

Clements ،Paul ،Rick Kazman ،and Mark Klein. (تقييم هيكلة البرمجيات: دراسة الحالة والمناهج)Evaluating Software Architectures: Methods and Case Studies. Boston ، MA: Addison-Wesley2002 ،.

Fowler ،Martin. (أنماط مشروع هيكلة التطبيقات)Patterns of Enterprise Application Architecture. Boston ،MA: AddisonWesley2002 ،.

Jacobson ،Ivar ،Grady Booch ،and James Rumbaugh. (عملية تطوير البرمجيات الموحدة)The Unified Software Development Process. Reading ،MA: Addison-Wesley1999 ،.

IEEE Std 1471-2000. (الأنشطة الموصى بها لوصف الهيكلة لأنظمة البرمجيات المكثفة)Recommended Practice for Architectural Description of Software Intensive Systems. Los Alamitos ،CA: IEEE Computer Society Press

هذا المستند هو دليل IEEE-ANSI لإنشاء مواصفات هيكلة البرمجيات.

[1] cc2e.com/0372

النُهج العامة لتطوير البرمجيات[1]:

العديد من الكتب المتاحة التي رسمت نُهج مختلفة لإدارة مشروع البرمجيات. بعضها أكثر تتابعاً، وبعضها الآخر أكثر إعادة وتكراراً.

.McConnell ،Steve. (دليل استمرار مشروع البرمجيات) Software Project Survival Guide. Redmond ،WA: Microsoft Press1998،

يقدم هذا الكتاب طريقة معينة لتنفيذ المشروع. ويؤكد النهج المقدم على التخطيط المدروس المسبق، وتطوير المتطلبات، وعمل الهيكلة يليها تنفيذ المشروع بعناية.

فهو يوفر إمكانية التنبؤ بعيدة المدى بالتكاليف والجداول الزمنية، والجودة العالية، وبقدر مقبول من المرونة.

Kruchten ،Philippe. (العملية المنطقية الموحدة)The Rational Unified Process: An Introduction2 ،d ed. Reading ،MA Addison-Wesley2000 ،.

يقدم هذا الكتاب نهج مشروع "الهيكلية المركزية وقيادة حالات الاستخدام"" architecture centric ".and use-case driven”

كما في دليل بقاء مشروع البرمجيات. فإنه يركز على العمل المسبق الذي يوفر إمكانية التنبؤ بعيدة المدى بالتكاليف والجداول الزمنية، والجودة العالية، وقدر مقبول من المرونة.

نهج هذا الكتاب يتطلب استخدام أكثر تمرساً بعض الشيء من المناهج الموصوفة في دليل استمرار مشروع البرمجيات وشرح البرمجة الفائقة: احتواء التغيير.

Jacobson ،Ivar ،Grady Booch ،and James Rumbaugh. (عملية تطوير البرمجيات الموحدة)The Unified Software Development Process. Reading ،MA: Addison-Wesley1999 ،.

هذا الكتاب أكثر عمقاً في معالجة العناوين المغطاة في"An Introduction، 2ed d ".

Beck ،Kent. (البرمجة الفائقة الموضّحة)Extreme Programming Explained: Embrace Change. Reading ،MA: Addison Wesley2000 ،.

ويصف "بيك" نهجاً تكرارياً للغاية يركز على تطوير المتطلبات والتصاميم بشكل متكرر، بالتزامن مع البناء. حيث يوفر نهج البرمجة الفائقة القليل من القدرة على التنبؤ بعيد المدى ولكنه يوفر درجة عالية من المرونة.

[1] cc2e.com/0379

Gilb ،Tom. (مبادئ إدارة هندسة البرمجيات)Principles of Software Engineering Management. Wokingham ،England: Addison-Wesley1988 ،.

ويستكشف نهج "غليب" التخطيط الحذر والمتطلبات، وقضايا الهيكلة في وقت مبكر من المشروع ومن ثم يوائم باستمرار خطط المشروع مع تقدمه (المشروع) ويوفر هذا النهج القدرة على التنبؤ بعيد المدى، وجودة عالية، ودرجة عالية من المرونة. ولكنه يتطلب تمرس أكثر من النهج الموضح في "بقاء مشروع البرمجيات" و"شرح البرمجة الفائقة: احتواء التغيير."

McConnell ،Steve. (التطوير السريع)Rapid Development. Redmond ،WA: Microsoft Press، 1996.

يقدم هذا الكتاب نهج صندوق أدوات للتخطيط للمشروع، يستطيع مُخطِط المشروع الخبير استخدام الأداة المقدمة في هذا الكتاب لإنشاء خطة مشروع المناسبة لمتطلبات مشروعه الخاصة.

Boehm ،Barry and Richard Turner. (موازنة بين خفة الحركة والانضباط: دليل التائه)Balancing Agility and Discipline: A Guide for the Perplexed. Boston ،MA: Addison-Wesley، 2003.

يستكشف هذا الكتاب التباين بين التطوير السريع وأنماط التطوير المعتمدة على التخطيط.

الفصل الثالث يحوي أربعة أقسام توضح بشكل خاص "يوم نموذجي باستخدام PSP/TSP"، "يوم نموذجي باستخدام البرمجة الفائقة"، "يوم حرج باستخدام PSP/TSP "، و"يوم حرج باستخدام البرمجة الفائقة"

الفصل الخامس وهو حول استخدام المخاطر لتحقيق التوازن، والذي يوفر توجيهات حاسمة للاختيار بين نهج السرعة أو نهج التخطيط المسبق.

الفصل 6، "الاستنتاجات"، هو أيضا متوازن جيداً ويقدم منظوراً واسعاً. الملحق E هو "منجم الذهب" من البيانات التجريبية على الأنشطة السريعة.

Larman ،Craig. (التطوير السريع والتكراري: دليل المدير)Agile and Iterative Development: A Manager's Guide. Boston ،MA: Addison Wesley2004 ،.

وهو مقدمة بحثية جيدة للمرونة، أسلوب التطوير التدرجي. تعطي ملاحظات لسكروم Scrum، والبرمجة الفائقة، والعملية الموحدة، وEvo.

قائمة التحقق: المتطلبات التحضيرية1

- هل حددت نوع مشروع البرمجيات الذي تعمل عليه وقمت بتخصيص مقاربتك بشكل مناسب؟
- هل المتطلبات محددة ومستقرة بما فيه الكفاية لابدء بالبناء؟ (راجع قائمة التحقق من المتطلبات للحصول على التفاصيل).
- هل الهيكلية محددة بشكل كاف لبدء البناء؟ (انظر لائحة اختبار الهيكلة لمزيد من التفاصيل).
- هل لديك مخاطر أخرى خاصة بمشروعك تم تحديدها، مثل أن البناء لا يتعرض لمخاطر أكثر من اللازم؟

[1] cc2e.com/0386

نقاط مفتاحية

- يتمثل الهدف الشامل للتحضير للبناء في الحد من المخاطر. تأكد من أن أنشطتك التحضيرية تقلل المخاطر، ولا تزيدها.
- إذا كنت ترغب في تطوير برامج عالية الجودة، يجب أن يكون الاهتمام بالجودة جزءاً من عملية تطوير البرمجيات من البداية إلى النهاية. الانتباه إلى الجودة في البداية لها تأثير أكبر على جودة المنتج من الاهتمام في نهاية المشروع.
- تثقيف المدراء وزملاء العمل حول عملية تطوير البرمجيات هو جزء من وظيفة المبرمج، بما في ذلك أهمية التحضير الكافي قبل بدء البرمجة.
- نوع المشروع الذي تعمل عليه يؤثر بشكل كبير على متطلبات البناء الأساسية - يجب أن تكون العديد من المشاريع تكرارية للغاية، ويجب أن يكون بعضها أكثر تسلسلاً.
- إذا لم يُحدّد تعريف جيد للمشكلة، فربما تحل المشكلة الخطأ أثناء البناء.
- إذا لم تُحدّد المتطلبات بشكل جيد، فقد تغيب عنك تفاصيل هامة للمشكلة. تتغير تكاليف المتطلبات من 20 إلى 100 مرة في المراحل التالية البناء عما كانت عليه في وقت سابق، لذلك تأكد من المتطلبات صحيحة قبل بدء البرمجة.
- إذا لم يتم تصميم الهيكلية بشكل جيد، فقد تقوم بحل المشكلة الصحيحة بطريقة خاطئة أثناء البناء. تكلفة التغيرات في الهيكلية تزيد مع ازدياد الشفرة البرمجية المكتوبة للهيكلة الخاطئة، لذلك تأكد من أن الهيكلية صحيحة أيضاً.
- افهم النهج الذي اتخذت لمتطلبات البناء الأساسية لمشروعك، واختر نهج البناء الخاص وفقاً لذلك.

الفصل الرابع

قرارات بناء مفتاحية

المحتويات

مواضيع ذات صلة

- المتطلبات الأولية التحضيرية: الفصل 3
- تحديد نوع البرامج التي تعمل عليها: الجزء 2.3
- كيفية تأثير حجم البرنامج على البناء: الفصل 27
- إدارة البناء: الفصل 28
- تصميم البرمجيات: الفصل 5 والفصول من 6 إلى 9

بمجرد التأكد من وضع الأساس المناسب للبناء، يتجه التحضير نحو المزيد من القرارات الخاصة بالبناء. ناقش الفصل 3 "قس مرتين، اقطع مرة: المتطلبات الأولية التحضيرية" البرنامج المكافئ لمخططات العمل ورخص البناء. قد لا يكون لديك القدرة على التحكم بشكل كبير بتلك التحضيرات لهذا كان تركيز ذلك الفصل على تحديد ما هو متوفر لديك للعمل عندما يبدأ البناء. يركز هذا الفصل على التحضيرات التي يتحمل مسؤوليتها بشكل مباشر أو غير مباشر المبرمجون الفرديون `individual programmers` والقيادات الفنية. ويناقش البرنامج المكافئ كيفية اختيار أدوات محددة لحزام الأداة وكيفية تحميل شاحنتك قبل توجهك إلى موقع العمل.

إذا شعرت بأنك قرأت سلفاً ما فيه الكفاية حول تحضيرات البناء، يمكنك التوجه إلى الفصل 5 "التصميم في مرحلة البناء".

1.4 اختيار لغة البرمجة

" يفتح التدوين الجيد المجال للتركيز على المشاكل الأكثر تقدماً من خلال تخفيف التفكير في العمل غير الضروري وهذا يزيد من القدرة الذهنية للسباق. قبل ظهور التدوين العربي عملية الضرب كانت صعبة وعملية التقسيم حتى للأعداد الصحيحة تستدعي أعلى القدرات الرياضية. ربما لا شيء في العالم الحديث سيثير دهشة عالم رياضيات يوناني أكثر من أن يعلم أن.. نسبة كبيرة من سكان أوروبا الغربية يستطيعون تنفيذ عملية القسمة للأرقام الكبيرة. هذه الحقيقة ستبدو له مستحيلة تماماً. قوتنا المعاصرة في سهولة حساب الكسور العشرية هي على الأغلب نتيجة إعجازية للاستكشاف التدريجي للترميز الكامل."

--- ألفريد نورث وايتهيد (فيلسوف وعالم رياضيات انكليزي)

لغة البرمجة التي يطبقها النظام يجب أن تكون مفيدة جداً بالنسبة لك منذ بداية العمل بالبناء وحتى نهايته. تظهر الدراسات أن اختيار لغة البرمجة يؤثر بعدة طرق على الإنتاجية وجودة الشفرة.

يكون المبرمجون أكثر إنتاجيةً عند استخدام لغة مألوفة أكثر منها عند استخدام أخرى غير مألوفة. البيانات المستمدة من نموذج تقدير كوكومو الثاني Cocomo II estimation model تظهر أن المبرمجين الذين يعملون على لغة يستخدمونها مسبقاً لمدة ثلاث سنوات أو أكثر هم أكثر إنتاجية ب 30% عن المبرمجين المساوين لهم في الخبرة ويستخدمون لغة جديدة (Boehm et al. 2000). وجدت دراسة سابقة لـ (أي بي إم) أن المبرمجين الذين يملكون خبرة واسعة مع لغة البرمجة كانوا أكثر إنتاجية بثلاث مرات من قليلي الخبرة (Walston and Felix 1977). (نموذج كوكومو الثاني أكثر حرصاً لعزل آثار العوامل الفردية والتي تأخذ بعين الاعتبار النتائج المختلفة من دراستين مختلفتين.)

يحقق المبرمجون الذين يعملون على لغات عالية المستوى؛ إنتاجية وجودة أفضل من الذين يعملون على لغات ذات مستوى أقل. لغات مثل "سمول توك Smalltalk، وجافا Java، وسي ++C++، وفيجوال بيسك Visual Basic" مشهود لها بالفضل في تحسين الإنتاجية، والدقة، والبساطة والوضوح من 5 إلى 15 مرة عن اللغات منخفضة المستوى مثل لغة التجميع أسيمبلي assembly وسي c ((Brooks 1987، Jones 1998، Boehm 2000. إنك توفر الوقت عندما لا تحتاج إلى الاحتفال في كل مرة تقوم عبارة مكتوبة باللغة سي بالعمل المفترض بها أن تقوم به. وعلاوة على ذلك، فإن اللغات ذات المستوى الأعلى هي أكثر تعبيراً من اللغات ذات المستوى الأدنى. حيث أن كل سطر من التعليمات البرمجية يقول أكثر. يظهر الجدول 1.4 النسب القياسية لعبارات المصدر في عدة لغات عالية المستوى منسوبة إلى الشفرة المكافئة في لغة سي. وتعني النسبة الأعلى أن كل سطر من الشفرة في اللغة المذكورة ينجز أكثر من كل سطر من الشفرة في لغة سي.

الجدول 1.4 النسب القياسية لعبارات لغات عالية المستوى منسوبة إلى الشفرة المكافئة في لغة سي

اللغة	المستوى المقابل بلغة C
سي (C)	1
سي ++ (C ++)	2.5
فورتران 95 (Fortran 95)	2
جافا (Java)	2.5
بيرل (Perl)	6
بايثون (Python)	6
سمول توك (Smalltalk)	6
مايكروسوفت فيجوال بيسك (Microsoft Visual Basic)	4.5

المصدر: مقتبس من تقدير تكاليف البرامج (جونس 1998)، برنامج تقدير التكلفة مع نموذج كوكومو الثاني(Boehm 2000)، و" مقارنة تجريبية لسبع لغات برمجة " (Prechelt 2000)

بعض اللغات أفضل من غيرها في التعبير عن المفاهيم البرمجية. يمكنك التشبيه بين اللغات الطبيعية كالإنكليزية ولغات البرمجة مثل جافا وسي++. في حالة اللغات الطبيعية، افترض عالمي اللغات سابير وورف (Sapir and Whorf) أن هناك علاقة بين قوة تعبير اللغة والقدرة على التفكير بأفكار معينة. فرضية سابير-وورف تقول بأن قدرتك على التفكير تعتمد على معرفة كلمات قادرة على التعبير عن الفكرة. إذا لم تكن تعرف كلمات لن تستطيع التعبير عن فكرتك ويمكن أنك لن تكون قادرا حتى على صياغتها (وورف 1956).

يمكن أن يكون المبرمجون بشكل مشابه متأثرين بلغاتهم. الكلمات المتاحة في لغة البرمجة للتعبير عن الأفكار البرمجية تحدد بالتأكيد كيف تعبر عن أفكارك وربما تحدد حتى ما هي الأفكار التي تستطيع التعبير عنها.

الدليل شائع على تأثير لغات البرمجة على أفكار المبرمجين. القصة النموذجية تشبه: " نحن نقوم بكتابة نظام جديد في سي++، لكن أغلب مبرمجينا لا يملكون الخبرة الكافية فيها. لقد أتوا وعندهم خلفية في لغة الفورتران. كتبوا شيفرة مجمّعة في سي++، لكنهم بالحقيقة كتبوا فورتران مقنّعة. قاموا بتوسيع سي ++ لتحاكي الملامح السيئة للفورتران (مثل تعليمات الانتقال والبيانات العامة ومتجاهلين مجموعة غنية من قدرات البرمجة غرضية التوجه في لغة سي++. " هذه الظاهرة قد تم الإبلاغ عنها في جميع أنحاء صناعة البرمجيات لسنوات عديدة (هانسون 1984، يوردون 1986أ).

أوصاف اللغات

تاريخ تطوير بعض اللغات مثير للاهتمام، وكذلك قدراتها العامة. فيما يلي وصف للغات الأكثر شيوعا المستخدمة اليوم.

ادا Ada

ادا هي لغة برمجة عامة، عالية المستوى تعتمد على الباسكال. طورت تحت رعاية وزارة الدفاع وهي مناسبة بشكل خاص لبرامج لوقت الحقيقي real-time والأنظمة المضمنة embedded systems. تؤكد ادا على استخراج البيانات وإخفاء المعلومات وتجبرك على التفريق بين الأجزاء العامة والخاصة لكل صف وحزمة. اختير ادا كاسم للغة تكريماً لعالم الرياضيات آدا لوفيلاس Ada Lovelace الذي يُعتبر أول مبرمج في العالم. الاستخدام الرئيسي اليوم ل "ادا" في المجال العسكري والفضاء وأنظمة الطيران.

لغة التجميع اسيمبلي Assembly

لغة التجميع أو المجمع هو نوع من اللغات منخفضة المستوى فيها كل عبارة تقابل تعليمة آلة واحدة. لأن العبارات تستخدم تعليمات آلة محددة، إن لغة التجميع محددة لمعالج معين على سبيل المثال محددة لوحدات المعالجة المركزية إنتل أو موتورولا. يعتبر المجمع كلغة من الجيل الثاني. يتجنبها معظم المبرمجون ما لم يكونوا يريدون زيادة حدود سرعة التنفيذ وحجم الشفرة.

سي C

سي هي لغة برمجة عامة، متوسطة المستوى مرتبطة أصلا بنظام التشغيل يونيكس UNIX ولديها بعض ميزات اللغة عالية المستوى مثل البيانات المهيكلة structured data، وتدفق التحكم المهيكل structured control flow، واستقلالية الجهاز ومجموعة غنية من المعاملات. إنها تدعى أيضا "لغة تجميع محمولة" لأنها تستخدم المؤشرات والعناوين على نطاق واسع، ولديها بعض بنى اللغة منخفضة المستوى مثل معالجة البتات bit manipulation وهي ضعيفة الكتابة.

طورت لغة سي في السبعينيات في مختبرات بيل. ولقد صممت أصلاً واستخدمت على DEC PDP-11 والتي فيها نظام التشغيل، ومترجم لغة سي وتطبيقات اليونكس جميعها مكتوبة باستخدام لغة سي. تم إصدار معيار ANSI في عام 1988 الذي أصدر لينظم كتابة الشفرة في لغة سي والذي تم تنقيحه في عام 1999. كانت سي المعيار الفعلي للحواسيب الصغيرة وبرمجة محطات العمل في الثمانينات والتسعينات.

سي ++ (C ++)

سي++ هي لغة غرضية التوجه مرتكزة على لغة سي، طورت في الثمانينات في مختبرات بيل. بالإضافة لكونها متوافقة مع لغة سي، تؤمن سي++ الصفوف، وتعدد الأشكال، والتعامل مع الاستثناءات، والقوالب، وتؤمن فحص نمط متين أكثر من سي. كما تؤمن مكتبة قياسية ضخمة وشاملة.

سي# (C#)

سي شارب (C#) هي لغة برمجة أغراض عامة، غرضية التوجه وطوّرت بيئة البرمجة من قبل مايكروسوفت بأسلوب بناء جمل شبيه ل سي، وسي++ وجافا، وتؤمن أدوات شاملة تساعد في التطوّير على منصة مايكروسوفت.

كوبول Cobol

كوبول هي لغة برمجة شبيهة بالإنكليزية والتي طوّرت أساساً في الفترة بين 1959 -1961 لاستخدامها من قبل وزارة الدفاع. تستخدم الكوبول بشكل رئيسي لتطبيقات الأعمال التجارية وبقيت واحدة من أكثر اللغات انتشاراً حتى اليوم. هي الثانية فقط من حيث الشعبية بعد الفيجوال بيزك (فيمان ودرايفر 2002). تحدثت الكوبول عبر السنوات لتشمل التوابع الرياضية وإمكانية غرضية التوجه. الاسم كوبول Cobol مختصر من أوائل أحرف "اللغة المشتركة الموجهة نحو الأعمال، COmmon Business-Oriented Language "

الفورتران Fortran

الفورتران هي أول لغة حاسوب عالية المستوى، قدمت الأفكار حول المتغيرات والحلقات عالية المستوى. كلمة فورتران اختصار ل "ترجمة الصيغة FORmula TRANslation ". طورت الفورتران بالأصل في الخمسينيات وشهدت العديد من التنقيحات الهامة، بما في ذلك فورتران77 في العام 1977، والتي أضافت عبارات الكتل المهيكلة if-then-else ومعالجة السلاسل المحرفية. أضافت فورتران 90 (أنماط، ومؤشرات، وصفوف ومجموعة شاملة من العمليات على المصفوفات) المحددة من قبل المستخدم. تستخدم الفورتران بشكل رئيسي في التطبيقات العلمية والهندسية.

جافا Java

جافا هي لغة غرضية التوجه مع قواعد شبيهة ل سي وسي ++ مطورة من قبل شركة صن للأنظمة المصغرة المحدودة Sun Microsystem، Incصممت لغة جافا لتعمل على أي منصة عن طريق تحويل شفرة جافا المصدرية إلى شفرة بايت (ثمانية) byte code، عندها تعمل في أي منصة من خلال بيئة تعرف بالآلة افتراضية. لغة جافا واسعة الانتشار في برمجة تطبيقات الوب

جافا سكريبت JavaScript

جافا سكريبت هي لغة نصية (scripting) مفسّرة والتي هي مرتبطة قليلاً بالجافا. تستخدم في المقام الأول للبرمجة من جانب الزبون client-side مثل إضافة توابع بسيطة وتطبيقات الانترنت على صفحات الوب.

بيرل Perl

بيرل هي لغة التعامل مع السلاسل تعتمد على ال سي والعديد من خدمات اليونيكس. تستخدم البيرل غالباً لمهام إدارة النظام مثل إنشاء تدوينات البناء النصية scripts فضلاً عن توليد التقارير ومعالجتها. كما تستخدم لإنشاء تطبيقات الوب مثل موقع Slashdot. بيرل Perl هي اختصار لغة الاستخراج العملي والتقرير Practical Extraction and.Report Language

بي اتش بي PHP

بي اتش بي هي لغة بناء النصوص مفتوحة المصدر مع قواعد مشابهة لبيرل، وبورن شل Shell Bourne، وجافا سكريبت وسي. تعمل ال بي اتش بي على جميع أنظمة التشغيل الرئيسية لتنفيذ وظائف تفاعلية من جانب المخدّم. ويمكن أن تكون مضمّنة في صفحات الوب للوصول إلى معلومات قاعدة البيانات الحالية. أتى الاختصار PHP في الأصل من الصفحة الرئيسية الشخصية Personal Home Page لكنه الآن اختصار لمعالج النص التشعبي Hypertext Processor.

بايثون Python

بايثون هي لغة مفسّرة تفاعلية غرضية التوجه تعمل في بيئات عديدة. استخدامها الأكثر شيوعاً لكتابة النصوص وتطبيقات الوب الصغيرة كما تحوي أيضا بعض الدعم لإنشاء برامج أكبر.

اس كيو إل SQL

اس كيو إل هي في الواقع لغة الاستعلام القياسية، والتحديث وإدارة قواعد البيانات العلائقية. اس كيو إل اختصار للغة الاستعلام المُهيكلة. على عكس اللغات الأخرى المدرجة في هذا القسم فان اس كيو إل هي "لغة تصريحية" مما يعني أنها لا تعرف تسلسل العمليات وإنما نتيجة لبعض العمليات.

الفيجوال بيسك VISUL BASIC

النسخة الأصلية من بيزك كانت لغة عالية المستوى طورت في كلية دارتموث في الستينيات. الاختصار بيزك BASIC أتى من شفرة التعليمات الرمزية متعددة الأغراض للمبتدئين Beginner's All-purpose Symbolic Instruction Code. فيجوال بيزك هي إصدار مرئي غرضي التوجه عالي المستوى من لغة البيزك طورت من قبل شركة المايكروسوفت والتي تم تصميمها أصلا لإنشاء تطبيقات مايكروسوفت ويندوز. وتم منذ ذلك الحين توسيعها لتدعم تخصيص تطبيقات سطح المكتب مثل مايكروسوفت أوفيس، إنشاء برامج الوب والتطبيقات الأخرى. يقول الخبراء أنه في بداية الألفية الجديدة كان المطورين المحترفين يعملون في لغة الفيجوال بيزك أكثر من أي لغة أخرى (فيمان ودرايفر 2002).

2.4 اتفاقيات البرمجة[1]

في البرمجيات عالية الجودة تستطيع أن ترى العلاقة بين التكامل المفاهيمي للهيكلة والتنفيذ منخفض المستوى. يجب أن يكون التنفيذ منسجما مع الهيكلة التي توجهه وتنسقه داخليا. وهذه هي النقطة في إرشادات البناء لأسماء المتغيرات، وأسماء الصفوف، وأسماء الإجرائيات، واتفاقيات التنسيق واتفاقيات التعليق.

في البرنامج المعقد تعطي الارشادات الهيكلية التوازن الهيكلي للبرنامج، وتؤمن إرشادات البناء التناغم في المستويات المنخفضة له، موضحةً كل صف كجزء مهم من التصميم الشامل. حيث يتطلب أي برنامج كبير بنية تحكم توحد تفاصيل لغة برمجته. فجزء من جمال البنية الكبيرة هو في الطريقة التي تحمل من خلالها أجزائها المفصلة بصمة هيكليتها. وبدون ضبط موحد فإن إنشائك سيكون مزيجا من الاختلاف السيئ في الأسلوب. مثل هذه الاختلافات ترهق ذهنك، وبسبب صعوبة فهم اختلافات تصميم الشفرة التي تكون اعتباطية، فإن أحد مفاتيح البرمجة الناجحة تكون بتجنب الاختلافات الاعتباطية بحيث يتمكن ذهنك من التركيز بحريه على لاختلافات التي تحتاجها حقا. للمزيد انظر "التقنيات الرئيسية الضرورية للبرمجيات: إدارة التعقيد في القسم 2.5.

ماذا لو كان لديك تصميم كبير للرسم. وكان هناك جزء كلاسيكي، وجزء انطباعي، وجزء ثلاثي الأبعاد؟ هذا لن يحقق وحدة المفاهيم بغض النظر عن مدى قربك من إدراك التصميم الكلّي. فهي تبدو مثل مجموعة من القطع المختلفة. يحتاج البرنامج أيضا إلى وحدة منخفضة المستوى.

قبل أن يبدأ البناء، وضح اتفاقيات البرمجة التي ستستخدمها. تفاصيل اتفاقية كتابة الشفرة هي في مثل هذا المستوى من الدقة التي يكاد يكون فيها من المستحيل تعديلها داخل البرنامج بعد أن تمت كتابتها. وترد تفاصيل هذه الاتفاقيات في جميع أنحاء الكتاب.

3.4 موقعك على الموجة التكنولوجية

خلال مسيرتي المهنية رأيت ارتفاع نجم الحواسب الشخصية في حين انحدر نجم الحواسب الرئيسية الكبيرة mainframes نحو الأفق. لقد رأيت برامج واجهة المستخدم الرسومية GUI programs تحل محل البرامج القائمة على المحارف. ورأيت الوب يصعد بينما ينخفض الويندوز. أستطيع فقط أن أفترض أنه في الوقت الذي تقرأ فيه هذا ستصعد بعض التكنولوجيا الجديدة، وبرمجة الوب كما أعرفها اليوم (2004) سوف تكون في طريقها للنهاية. دورات للتكنولوجيا هذه، أو الموجات، تتضمن أنشطة برمجة مختلفة اعتماداً على المكان الذي تجد نفسك فيه على الموجة.

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول قوة الاتفاقيات، انظر الأقسام من 11.3 إلى 11.5.

في بيئات التكنولوجيا الناضجة- نهاية الموجة، مثل برمجة الوب في منتصف العقد الأول من الألفية الجديدة - نستفيد من البنية التحتية الغنية لتطوير البرمجيات. توفر بيئات الموجة الأخيرة العديد من خيارات لغة البرمجة، فحص شامل للخطأ للشفرة المكتوبة في تلك اللغات، أدوات قوية للمصحح، وتحسين الأداء الأمثل الموثوق والأوتوماتيكي. المترجمات تقريباً خالية من الأخطاء. وتُوثق الأدوات بشكل جيد في كتب البائع "المزود" وفي كتب ومقالات الطرف الثالث، وفي موارد الوب الشاملة. يتم دمج الأدوات، لذلك يمكنك أن تقوم بعمل واجهة المستخدم، وقاعدة البيانات، والتقارير، والمنطق التجاري من خلال بيئة واحدة.

إذا واجهتك المشاكل، يمكنك بسهولة العثور على الحيل من الأدوات الموضحة في الأسئلة الشائعة العديد من الاستشاريين ودروس التدريب متاحة أيضاً.

في بيئات الموجات الأقدم - على سبيل المثال برمجة الويب في منتصف التسعينات- الوضع هو العكس. خيارات قليلة متاحة للغة البرمجة، وتلك اللغات تميل إلى أن تكون مليئة بالأخطاء وموثقة بشكل ضعيف.

يقضي المبرمجين الكثير من الوقت لمجرد محاولة معرفة كيفية عمل اللغة بدلاً من كتابة شفرة جديدة. كما يقضي المبرمجون ساعات لا تحصى من العمل حول الأخطاء في منتجات اللغة، ونظام التشغيل الأساسي، وغيرها من الأدوات.

تميل أدوات البرمجة في بيئات الموجات الأقدم إلى أن تكون بدائية. قد لا تكون المصححات موجودة على الإطلاق، ومحسنات المترجم لا تزال فقط كوميض في أعين بعض المبرمجين. يقوم البائعون في كثير من الأحيان تعديل إصدار المجمع الخاص بهم، ويبدو أن كل إصدار جديد ينتهك أجزاء هامة من شفرتك. الأدوات ليست متكاملة، لذلك تميل إلى العمل مع أدوات مختلفة لواجهة المستخدم، وقاعدة البيانات، والتقارير، ومنطق الأعمال.

لا تميل الأدوات إلى أن تكون متوافقة بشكل كبير، ويمكنك أن تنفق قدراً كبيراً من الجهد فقط للحفاظ على عمل الوظائف الموجودة لمواجهة هجوم إصدارات المترجم والمكتبات.

إذا واجهتك مشكلة، توجد الأدبيات المرجعية على الوب في شكل ما، ولكنها ليست موثوقة دائماً، وإذا كان مرجعك المتاح هو أي دليل، ففي كل مرة تواجه مشكلة يبدو كما لو كنت أول شخص سيقوم بذلك.

وقد تبدو هذه التعليقات كأنها توصية لتجنب برمجة الموجات المبكرة، ولكن هذا ليس القصد منها. بعض التطبيقات الأكثر ابتكاراً تنشأ من برامج الموجات المبكرة، مثل توربو باسكال Turbo Pascal، لوتُس 123 Lotus 123، مايكروسوفت وورد Microsoft Word، ومستعرض الموزاييك the Mosaic browser.

النقطة هي أنه كيفية قضائك لأيام البرمجة الخاصة بك سوف تعتمد على موقعك على موجة التكنولوجيا إذا كنت في الجزء الأخير من الموجة، يمكنك أن تخطط أنك ستقضي جزءاً كبيراً من يومك وأنت تكتب التوابع الجديدة بانتظام.

إذا كنت في الجزء الأول من الموجة، يمكنك أن تفترض أنك ستأخذ حصة كبيرة من وقتك في محاولة لمعرفة الميزات غير الموثقة للغة البرمجة الخاصة بك، وأخطاء التصحيح التي تتحول لتصبح عيوبا في شفرة المكتبة، وتنقيح الشفرة بحيث أنها ستعمل مع إصدار جديد من مكتبة بعض البائعين، وهكذا.

عندما تجد نفسك تعمل في بيئة بدائية، تدرك أن أنشطة البرمجة الموصوفة في هذا الكتاب يمكن أن تساعدك أكثر مما يمكن في البيئات الناضجة. كما أشار ديفيد غريز، أدوات البرمجة الخاصة بك لا يجب أن تحدد كيف تفكر حول البرمجة (1981). ميّز غريز بين البرمجة في لغة مقابل البرمجة إلى لغة. المبرمجين الذين يبرمجون "في" لغة يحدّون أفكارهم في البنى التي تدعمها اللغة بشكل. إذا كانت أدوات اللغة بدائية، فإن أفكار المبرمج ستكون أيضا بدائية. يقرر المبرمجون الذين يبرمجون "إلى" لغة أولاً ما الأفكار التي يريدون أن يعبروا عنها، ثم يحددون كيفية التعبير عن تلك الأفكار باستخدام الأدوات التي توفرها لغتهم المحددة.

أمثلة عن البرمجة إلى لغة

في الأيام الأولى من الفيجوال بيزك، كنت محبطا لأنني أردت الحفاظ على كل من منطق الأعمال، وواجهة المستخدم، وقاعدة البيانات بشكل منفصل في المنتج الذي أطوره، لكن لم يكن هناك أية طرق مضمّنة في اللغة لفعل ذلك. لقد عرفت بأنني إذا لم أكن حذراً، فإنه مع مرور الوقت بعض "إطارات" الفيجوال بيزك ستحتوي على منطق الاعمال، بعض الإطارات ستحتوي على شفرة قاعدة البيانات، وبعضها لن يحتويها – وأنا لن أكون قادراً في نهاية المطاف أن أتذكر مكان كل شفرة. كنت قد أكملت للتو مشروع سي ++ الذي قام بعمل ضعيف في فصل هذه القضايا، وأنا لا أريد أن أجرب شيئا من الصداع الذي سبقت تجربته في لغة أخرى.

بناء على ذلك، لقد اعتمدت على اتفاقية تصميم بأن ملف frm. (ملف النافذة) سمح له فقط استعادة البيانات من قاعدة البيانات وتخزين البيانات مرة أخرى في قاعدة البيانات. لم يكن مسموحاً اتصال تلك البيانات بشكل مباشر مع الأجزاء الأخرى من البرنامج. كل إطار دعمت إجرائية ()IsFormCompleted، والذي استخدمه إجرائية الاتصال "calling routine" لتحديد ما إذا كان الإطار التي تم تنشيطها قد حفظت بياناتها. ()IsFormCompleted هي الإجرائية العامة الوحيدة التي تسمح للإطارات بذلك. لم يكن مسموحا للنوافذ بأن تحتوي على أي منطق للأعمال. كل الشفرات الأخرى يجب أن تحتوي ملف bas. المترابط، متضمنة التحقق من صحة الإدخالات في الإطار.

لم تشجع الفيجوال بيزك هذا النوع من النهج. وشجعت المبرمجين لوضع أكبر قدر ممكن من الشفرة في ملف frm.، ولم تجعل من السهل لملف frm. أن يستدعى إلى ملف bas. المرتبط.

هذه الاتفاقية بسيطة جداً، لكن كلما تعمقت في مشروعي، وجدت أنها تساعدني على تجنب العديد من الحالات التي كنت فيها أكتب شفرة معقدة من دون الاتفاقية. كنت أحمل الإطارات لكن أبقيهم مخفيين كي أستطيع استدعاء إجرائيات التحقق من صحة البيانات الموجودة فيهم، أو كنت أنسخ الشفرة من الإطارات إلى مواقع أخرى ومن ثم أقوم بصيانة الشفرة الموازية في أماكن متعددة. اتفاقية `IsFormCompleted()` أبقت الأشياء بسيطة أيضاً. لأن كل إطار يعمل بالضبط بنفس الطريقة، لا يجب على أن أخمن ثانية دلالات `IsFormCompleted()` – هذا يعني أن نفس الشيء سيحدث في كل مرة تُستخدم.

لم تدعم الفيجوال بيزك هذه الاتفاقية بشكل مباشر، لكن استخدامي لاتفاقية برمجة بسيطة – البرمجة إلى اللغة- صنع من أجل ضعف هيكلة اللغة في ذلك الوقت وساعد على إبقاء المشروع قابلا للإدارة بشكل معقول.



فهم الفارق بين البرمجة في لغة والبرمجة إلى لغة هو أمر بالغ الأهمية لفهم هذا الكتاب. معظم المبادئ البرمجية الهامة لا تعتمد على لغة محددة لكن على الطريقة التي تستخدمهم بها. إذا افتقرت لغتك للهياكل التي تريد استخدامها أو كانت عرضة لأنواع أخرى من المشاكل، حاول أن تعوضهم. اخترع اتفاقيات كتابة الشفرة الخاصة بك، والمعايير، ومكتبات الصف، وزيادات أخرى.

4.4 اختيار أنشطة البناء الرئيسية

يحدد جزء من الإعداد للبناء أي من الأنشطة الجيدة المتاحة التي ستحققها. بعض المشاريع تستخدم زوج البرمجة وتطوير "الاختبار أولاً"، بينما يستخدم الآخرون التطوير الأحادي وعمليات التفحص الرسمية. أيا من دمج التقنيات المذكورين سابقاً يمكن أن يعمل جيداً تبعاً لظروف المشروع المحددة.

تلخص قائمة التحقق الأنشطة المحددة التي يجب عليك تقرير تضمينها أو استبعادها خلال البناء. تفاصيل هذه الأنشطة موجودة في جميع أنحاء الكتاب.

لائحة اختبار: نشاطات البناء الرئيسية

كتابة الشفرة

- هل عرّفت كم من التصـميم سـيكون على الواجهة وكم سـيكون على لوحة المفاتيح، عندما تتم كتابة الشفرة؟
- هل عرّفت اتفاقيات الشفرة من أجل الأسماء، والتعليقات والتخطيط.

- هل عرّفت أنشطة الشفرة المحددة المضمنة في الهيكلة، مثل كيفية التعامل مع ظروف الخطأ، كيفية معالجة الأمن، ما هي الاتفاقيات التي ستستخدم لواجهات الصف، ما هي المعايير التي ستطبق الشفرة المعاد استخدامها، كم يؤخذ الأداء بالاعتبار أثناء كتابة الشفرة، وما إلى ذلك؟
- هل حددت موقعك على الموجة التكنولوجية وضبطت نهجك الخاص لتواكبها؟ هل حددت عند الضرورة كيف ستبرمج إلى اللغة بدلا من أن تكون محدودا بالبرمجة فيها؟

فريق العمل

- هل عرّفت إجراء تكامل – أي، هل عرّفت الخطوات المحددة التي على المبرمج أن يمر بها قبل التحقق من الشفرة داخل المصادر الرئيسة؟
- هل يبرمج المبرمجين في أزواج، أو بشكل فردي، أو هو مزيج من الاثنين؟

تأكيد الجودة

- هل سيكتب المبرمجون حالات اختبار test cases لشفراتهم قبل كتابة الشفرة نفسها؟
- هل سيكتب المبرمجون اختبارات الوحدة unit tests لشفراتهم بغض النظر عما إذا قاموا بكتابتها أولاً أو أخيراً؟
- هل سيتقدم المبرمجون بشفرتهم في المصحح قبل أن يفحصوها؟
- هل سيقوم المبرمجون باختبار التكامل لشفرتهم قبل أن يفحصوها؟
- هل سيراجع أو سيفحص المبرمجون شفرات بعضهم البعض؟

أدوات

- هل قمت باختيار أداة التحكم بعناية؟
- هل قمت باختيار اللغة وإصدار اللغة أو إصدار المترجم compiler؟
- هل قمت باختيار منصة مثل J2EE أو Microsoft.NET أو قررت صراحةً عدم استخدام إطار عمل؟
- هل قررت فيما إذا كنت ستسمح باستخدام ميزات اللغة غير القياسية؟
- هل حددت واكتسبت أدوات أخرى ستستخدمها مثل المحرر، وأدوات إعادة التصنيع، والمصحح، ومنصة الاختبار، والمدقق النحوي، وما إلى ذلك؟

نقاط مفتاحية

- لكل لغة برمجة نقاط قوة ونقاط ضــعف. كن على علم بنقاط القوة والضــعف المحددة في اللغة التي تستخدمها.
- ضـع اتفاقيات البرمجة قبل أن تبدأ بالبرمجة. من شـبه المسـتحيل تغيير الشـفرة لاحقاً لتناسـب هذه الاتفاقيات.
- توجد أنشطة بناء أكثر مما يمكنك استخدامه في أي مشروع. اختر بوعي الأنشطة الأنسب لمشروعك.
- اسأل نفسك عما إذا كانت الأنشطة البرمجية التي تستخدمها متوافقة مع لغة البرمجة التي تستخدمها أو تتحكم بواسطتها. تذكر أن كيف ستبرمج إلى اللغة بدلا من أن تكون محدودا بالبرمجة فيها؟
- موقعك على الموجة التكنولوجية يحدد ما هي النُهج التي ســتكون فعّالة– أو حتى ممكنة. حدد أين تكون على الموجة التكنولوجية، واضبط خططك وتوقعاتك وفقا لذلك.

القسم الثاني: إنشاء شفرة عالية الجودة

في هذا القسم:

الفصل 5

التصميم في البناء

المحتويات

مواضيع ذات صلة

- هيكلة البرمجية: القسم 3. 5
- صفوف جيدة: الفصل 6
- مميزات الإجرائيات عالية الجودة: الفصل 7
- البرمجة الوقائية: الفصل 8
- إعادة التصنيع: الفصل 24
- كيف يؤثر حجم البرنامج على البناء: الفصل 27

يجادل البعض حول كون التصميم يعتبر نشاط بناء حقاً، لكن عدّة أنشطة في المشاريع الصغيرة يُعتقد بأنّها بناء، وهي عادةً تتضمن التصميم. في مشاريع أكبر، ربما تقع الهيكلة الرسميّة على القضايا في مستوى النظام فقط وتترك عمل تصميمي أكبر لمرحلة البناء بشكل مقصود. وفي مشاريع أخرى كبيرة، قد يكون النظام قصداً مفصلاً بكفاية كي تصبح كتابة الشفرة عملاً مريحاً بشكل تام، ولكن نادراً ما يكون التصميم بهذا الكمال-عادةً، يصمم المبرمج جزء من البرنامج، سواء أكان رسمياً أو غير ذلك.

يتم الكثير من التصميم في المشاريع الصغيرة وغير الرسميّة عندما يجلس المبرمج أمام لوحة المفاتيح.[1] ربما يكون "التصميم" مجرد كتابة شفرة زائفة لواجهات الصفوف قبل كتابة التفاصيل. وربما يكون رسم مخططات لعلاقات بين صفوف قليلة قبل كتابة الشفرة. وقد يكون بسؤال مبرمج آخر أي نموذج تصميم يبدو أفضل خيار. بغض النظر عن كيفيّة إتمام التصميم، تستفيد المشاريع الصغيرة من التصميم الدقيق تماماً كالمشاريع الأكبر، وتعريف التصميم على أنه نشاط مستقل يزيد من الاستفادة الممكنة منه.

التصميم عنوان كبير لذلك فإنّ القليل فقط من مفاهيمه أُخذت بعين الاعتبار في هذا الفصل. يُحدد قسم كبير من التصميم الجيد للصفوف Classes والإجرائيات Routines من قبل هيكلة النظام، لذلك تأكّد من أنّ متطلب الهيكلة الذي نوقش في القسم 3. 5 حُقق. وحتى القيام بتصميم أكثر في مستوى الصفوف والاجرائيات المنفصلة شُرح في الفصل 6 "صفوف جيدة" والفصل 7 "إجرائيات عالية الجودة"

إذا كانت لديك معرفة بمواضيع تصميم البرمجيات، ربما ستقرأ العناوين العريضة في أجزاء تحدّيات التصميم في القسم 5-1 ومساعدات رئيسيّة على الاكتشاف في القسم 5-3

5-1 تحديات التصميم[2]

تعني العبارة "تصميم البرمجية" توليد أو اختراع أو ابتكار مخطط لتحويل مواصفات "برمجيّة حاسوب" إلى برنامج تنفيذي. التصميم هو النشاط الذي يربط المتطلبات بكتابة الشفرة والتصحيح. يؤمن التصميم الجيد للأشياء الأكثر أهمية بنية تمكن بأمان احتواء عدة تصاميم لأشياء أقل أهمية. التصميم الجيد مفيد في المشاريع الصغيرة ولا بد منه في المشاريع الكبيرة.

يواجه التصميم عدة تحدّيات، والتي حُدّدت خطوطها العريضة في هذا القسم.

التصميم هو مشكلة عويصة[3]

عرّف هورس رتل و ميلفن ويبر "المشكلة "العويصة" بأنها التي يمكن أن تُعرّف بشكل واضح فقط بحلها، أو حل جزء منها (1973). هذه المفارقة تقتضي أساساً أنّه عليك أن "تحل" المشكلة مرة كي تعرفها بشكل واضح

---

[1] **إشارة مرجعية** لتفاصيل عن المستويات المختلفة للشكلانية (formality) المطلوبة في المشاريع الصغيرة والكبيرة، راجع الفصل 27، "كيف يؤثر حجم المشروع على البناء"

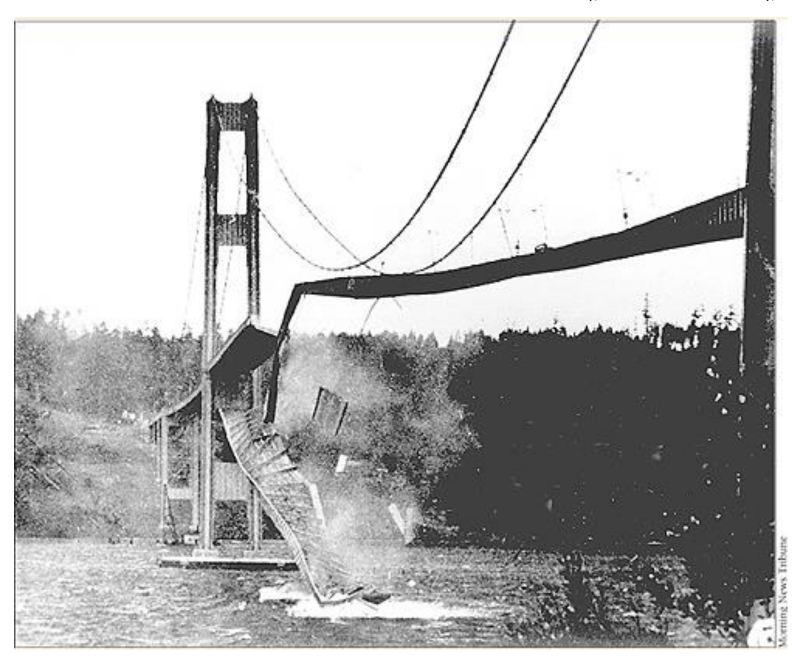
[2] **إشارة مرجعية** الفرق بين العملية المساعدة على الاكتشاف والعملية الحتمية مشروح في الفصل 2، "الاستعارات لفهم أفضل لتطوير البرمجيات"

[3] الصورة التي ترى فيها مصمم البرمجيات يطور تصميمه من وثيقة المتطلبات في أسلوب منطقي وخال من الخطأ هي حتماً صورة خيالية، لا يوجد أي نظام طُور بتلك الطريقة، وربما لن يوجد على الاطلاق. حتى تطوير البرامج الصغيرة الموجود في الكتب التخصصية والمقالات الاكاديمية غير واقعي. لقد نُقحت ولُمعت حتى أرانا الكاتب ما يتمنى أن يكون قد صنع، وليس ما حدث بالضبط. -ديفيد بورناس وباول كليمينتس.

وتحلها ثانية لتبتكر الحل الناجح. هذه العملية كانت الشعار العالمي في مجال تطوير البرمجيات لعقود (بيتيرز و ترب 1976).

في جعبتي عن هذا العالم مثال فظيع عن مشاكل كهذه، وهو التصميم الأول لجسر "تاكوما ناروز". فعندما كان يُبنى الجسر، كان الاهتمام الرئيسي في التصميم بأن يكون الجسر قوياً كفاية ليتحمل الأثقال المقدرة. في خصوصيّة جسر تاكوما ناروز، أنشأت الريح تموجات أفقية غير متوقعة. وفي يوم عاصف عام 1940، كبرت التموجات بشكل خارج عن السيطرة حتى انهار الجسر، كما هو موضح بالشكل 5-1.

هذا مثال جيد للمشكلة العويصة لأنه، حتى انهيار الجسر، لم يعرف المهندسون القائمون عليه أنّ ديناميكية الهواء يجب أن تؤخذ بعين الاعتبار في سياقٍ كهذا. فقط ببناء الجسر (حل المشكلة) استطاعوا أن يتعلموا عن الاعتبارات الإضافيّة في المشكلة والتي سمحت لهم بأن يبنوا جسر آخر لايزال صامداً حتى الآن.



الشكل 5-1 جسر تاكوما ناروز-مثال على المشكلة العويصة

أحد الاختلافات الرئيسية بين البرامج التي تطورها في المدرسة والتي تطورها في المهنة هو ندرة المشاكل العويصة –إن وجدت- عند حل مشاكل التصميم في البرامج المدرسية. تُصمم الوظائف المدرسية لتنقلك في خط مستقيم من البداية إلى النهاية. ربما تريد أن تعاقب بالريش والقطران (من أشد العقوبات المدرسية الامريكية) المدرّس الذي أعطاك وظيفة برمجية، وغيّرها عندما انتهيتَ من التصميم، ثم عندما انتهيتَ تقريباً من البرنامج كاملاً عاد وغيّرها. لكن هذه العملية بالذات هي واقع نعيشه يومياً في البرمجة المهنية.

التصميم عملية غير أنيقة (حتى لو جاءت بنتائج أنيقة)

يجب أن يبدو التصميم النهائي مرتباً ونظيفاً بشكل جيد، لكن العملية المستخدمة لتطوير التصميم ليست قريبة من أناقة النتيجة النهائية.

التصميم غير أنيق لأنك تخطو عدّة خطوات خاطئة وتمشي في عدة طرق مسدودة-ستصنع الكثير من الأخطاء[1]. في الواقع، ارتكاب الأخطاء هو الغاية من التصميم-أن ترتكب الأخطاء وتصحّح التصاميم أقل كلفةً من أن ترتكب نفس الأخطاء وتكتشفها بعد كتابة الشفرة، وعندها سيكون عليك أن تصحّح شفرة ناضجة. التصميم غير أنيق لان التصميم الجيد عادة ما يختلف بأمور دقيقة عن التصميم الرديء

التصميم غير أنيق أيضاً لأنّه من الصعب أن تعرف متى يكون التصميم "جيداً كفاية.[2]" كم هي التفاصيل كافية؟ ما هي كمية التصميم الذي يجب أن ينجز بترميز التصميم الرسمي، وما هي الكمية التي يجب أن تترك لتنجز باستخدام لوحة المفاتيح؟ متى تنتهي؟ طالما أن التصميم ذو نهاية مفتوحة، فالجواب الأكثر شيوعاً لهذا السؤال هو "عندما ينفذ منك الوقت."

التصميم يختص بالتبادلات والأولويات

في العالم المثالي، كل نظام يمكن أن يقلع فوراً، ويستهلك صفراً من مساحة التخزين، ويستخدم صفر من عرض حزمة الشبكة، ولا يحتوي أي أخطاء، ويكلف لا شيء كي ينشأ. أساس عمل المصمم في العالم الحقيقي أن يزن مميزات التصميم المتنافسة ويجد طريقة ليكون عادلاً بين هذه المميزات، إذا كانت سرعة معدل الاستجابة أهم من تصغير زمن التطوير، سيختار المصمم تصميماً ما. أما إذا كان تصغير وقت التطوير أهم، فسيختار المصمم الجيد بمهارة تصميماً آخر.

التصميم يتضمن التقييدات

إنّ إيجاد الاحتمالات غاية جزئية للتصميم وكذلك تقييد الاحتمالات. إذا امتلك النّاس وقت وموارد ومساحة لا نهاية لها وطُلب منهم تشييد منشآت مادية، لرأيت منشآت مبعثرة بشكل غير معقول وكل منشأة فيها غرفة لكل حذاء ومئات الغرف. هكذا يمكن أن يصبح حال البرمجية بدون تقييدات مفروضة عمداً. قيد "الموارد المحدودة" في تشييد الأبنية يجبر على تبسيط الحل والذي بالنهاية يحسّن الحل نفسه. الهدف في تصميم البرمجية هو ذاته.

التصميم غير حتمي

إذا أرسلت ثلاثة أشخاص بعيداً ليصمموا نفس البرنامج، يمكن ببساطة أن يعودوا مع ثلاثة تصاميم مختلفة كثيراً، وكل منها يمكن أن يكون مقبول تماماً. ربما يوجد أكثر من طريقة لسلخ هرة، لكن يوجد عادة دزينات من الطرق لتصمم برنامج حاسوبي.

[1] **اقرأ أيضاً** لبحث أكمل في هذه النقطة، راجع "عملية تصميم منطقية: سبب وكيفية تزييفها" (بارناس و كليمينفس 1986)

[2] **إشارة مرجعية** لجواب أفضل عن هذا السؤال، راجع "إلى أي مدى التصميم كاف؟" في القسم 5-4 لاحقاً في هذا الفصل.

التصميم هو عملية استكشافية



لكون التصميم غير حتمي، تميل تقنيات التصميم لتكون استكشافية-"قاعدة الإبهام" أو "التجريب أشياء نجحت في بعض الأحيان"-بدلاً من أن تكون عمليات قابلة للتكرار والتي تضمن أن تقدّم نتائج يمكن توقعها. التصميم يتضمن "التجريب والمحاكمة" والخطأ. أداة التصميم التي نجحت في عمل أو جانب ما من عمل قد لا تنجح في المشروع القادم. لا يوجد أداة صالحة لكل شيء.

التصميم أمر تراكمي[1]

لألخص بأناقة خصائص التصميم هذه أقول: التصميم "أمر تراكمي." التصاميم لا تنبثق فوراً من دماغ شخص ما بتكوين تام. إنها تتطوّر وتتحسّن من خلال مراجعات التصميم والمناقشات غير الرسمية وتجريب كتابة الشفرة نفسها وتجريب تنقيح الشفرة.

تخضع كل الأنظمة بشكل افتراضي إلى درجة ما من التغيرات في التصميم في بداية التطوير، ثم تنتقل (الأنظمة) بشكل قياسي إلى مجال أكبر كلما تمددت إلى إصدارات لاحقة[2]. الدرجة التي تحدّد مدى فائدة التغيير وقبوله تعتمد على طبيعة البرمجية تحت البناء.

2.5 مفاهيم التصميم المفتاحية

التصميم الجيد يعتمد على فهم مجموعة من المفاهيم الأساسية. يناقش هذا القسم دور التعقيد، وخصائص التصاميم المرغوبة، ومستويات التصميم.

الإلزام التقني الرئيسي للبرمجية: إدارة التعقيد

لفهم أهمية إدارة التعقيد، من المفيد مراجعة مقالة فريد بروكس الهامّة، "لا طلقات فضيّة: جوهر وأعراض هندسة البرمجيات" (1987) "No Silver Bullets: Essence and Accidents of Software Engineering" (1987).[3]

---

[1] cc2e.com/0539

[2] **اقرأ أيضاً** ليست البرمجية النوع الوحيد من المنشآت التي تتغير عبر الزمن. المنشآت المادية تتطور أيضا—انظر كيف تتعلم الأبنية (براند 1995) (see How Buildings Learn (Brand 1995).

[3] إشارة مرجعية لمناقشة طريقة تأثير التعقيد على نشاطات البرمجة ما عدا التصميم، راجع القسم 34-1، "التغلب على التعقيد."

الصعوبات الجوهرية والعرضية

افترض بروكس أن تطوير البرمجية يواجه صعوبات بسبب نوعين مختلفين من المشاكل-الجوهري والعرضي. بالحديث عن هذين المصطلحين، اعتمد بروكس على تقليد فلسفي يعود إلى ارسطو. في الفلسفة، الخصائص الجوهرية هي التي يجب حتماً أن يمتلكها الشيء ليكون ذلك الشيء. يجب حتماً أن تمتلك السيارة محرك وعجلات وأبواب لتكون سيارة. إذا لم تمتلك أيّا من تلك الخصائص الجوهرية، فهي بالحقيقة ليست سيارة.

الخصائص العرضيّة هي التي حدث وامتلكها شيء، هي الخصائص التي لا تعطي حكماً بكون الشيء هو. يمكن أن تمتلك السيارة محرك ف8 أو مزود بمسرع و4-أسطوانات أو أي نوع آخر من المحركات وتكون سيارة بعض النظر عن التفاصيل تلك. يمكن أن تمتلك السيارة بابين أو أربعة، يمكن أن تمتلك عجلات قليلة العرض أو عجلات من المغنيزيوم. كل هذه التفاصيل خصائص عرضيّة. يمكن أن تعتقد أنّ بالخصائص العرضيّة هي تصادفية أو تابعة لهوى الشخص أو اختيارية أو ممكنة الحدوث.

لاحظ بروكس أنّ الصعوبات العرضية الرئيسية في البرمجيات تُعالج منذ وقت طويل[1]. على سبيل المثال، الصعوبات العرضيّة المتعلقة بقواعد اللغة غير المتقنة تم استئصالها على نطاق واسع بالتطوّر من لغة المجمع إلى لغات الجيل الثالث ويتم إلغاؤها (الصعوبات) بشكل ملحوظ ومتزايد منذ ذلك الوقت. الصعوبات العرضيّة المتعلقة بالحواسيب غير التفاعلية حُلّت عندما حلت أنظمة "التشارك في الوقت" محل الأنظمة الدفعيّة. أقصت بيئات البرمجة المدمجة بعيداً عدم الفاعلية في العمل البرمجي الناشئ من أدوات تعمل مع بعضها برداءة.

ناقش بروكس فكرة أن التقدم في معالجة الصعوبات الجوهرية المتبقية للبرمجيات مقيد بأن يكون أبطأ. السبب هو أن التطوير البرمجي، بجوهره، يتألف من استنباط كل التفاصيل لمجموعة معقدة جداً ومتشابكة من المفاهيم. تظهر الصعوبات الجوهرية من ضرورة مواجهة واقع معقد وفوضوي؛ وتعريف الاعتمادات والحالات الشاذة بدقّة وكمال؛ وتصميم الحلول التي لا يمكن أن تكون تقريبا صالحة لكن يجب حتماً أن تكون صالحة تماماً؛ وما إلى ذلك. حتى لو تمكنا من اختراع لغة برمجة تستخدم نفس الاصطلاحات المستخدمة في مشاكل الواقع التي نحاول حلها. ستبقى البرمجة صعبة بسبب التحدي الكامن في تحديد كيف يعمل الواقع بدقّة. بما أنّ البرمجيات تعالج مشاكل واقعيّة متزايدة دوماً، والتفاعلات بين كينونات الواقع تصبح بشكل متزايد أعقد، وذلك بدوره يزيد الصعوبة الجوهرية في الحلول البرمجية.

الأصل لكل هذه الصعوبات الجوهرية هو التعقيد-العرضي والجوهري كلاهما.

[1] **إشارة مرجعية** تبرز الصعوبات العرضية في "الموجة المتقدمة" للتطوير أكثر من الموجة المتأخرة. لتفاصيل أكثر، راجع القسم 4-3، "مكانك على الموجة التقنية"

أهمية إدارة التعقيد[1].

عندما تعطي دراسات المشاريع البرمجية تقاريراً حول أسباب فشل المشاريع، نادراً ما يعرّفون الأسباب التقنية كمسبب رئيسي لفشل المشروع تفشل المشاريع بالغالبية العظمى بسبب المتطلبات الرديئة أو التخطيط الرديء أو الإدارة الرديئة. لكن عندما تفشل المشاريع حقاً لأسباب تقنية أساساً، يكون السبب غالباً هو التعقيد غير الخاضع للتحكم. يُسمح للبرمجية بأن تنمو إلى مستوى عالٍ من التعقيد بحيث لا أحد يعرف بحق ماذا تفعل. عندما يصل المشروع إلى نقطة لا أحد يفهم بشكل كامل أثر تغير الشفرة -في مكان ما- على الأماكن الأخرى؛ سيتباطأ التقدم حتى يتوقف تماماً.



إدارة التعقيد هي الموضوع التقني الأهم في تطوير البرمجيات. من وجهة نظري، من المهم جداً أن تكون إدارة التعقيد هي الإلزام التقني الرئيسي للبرمجيات.

أضف إلى معلوماتك" order of magnitude " "طبقات الحجم" نظام لترتيب الأشياء في طبقات حسب الحجم، بحيث كل طبقة أعلى من سابقتها بنسبة ثابتة (عادة 10). مثلاً "قيمة أكبر باثنتان من طبقات الحجم" يعنى قيمة أكبر بمئة مرة)

"مستويات الدلالة" في المنظور الحاسوبي البت له دلالتان، و910 بت لها عدد مستويات دلالة فظيع (2 مرفوع لقوة 910) ليس التعقيد صفة جديدة في تطوير البرمجيات. أشار الرائد في المجال الحاسوبي "إيدجر ديكسترا" إلى أن الحوسبة هي المهنة الوحيدة التي يلتزم فيها عقل واحد بأن يبني المساحة من بت إلى عدة مئات من الميغابايت، بنسبة 1 إلى 910، أو تسعة من "طبقات الحجم" "order of magnitude" (ديكسترا 1989). هذه النسبة الهائلة مذهلة. كتب ديكسترا "بالمقارنة إلى عدد "مستويات الدلالة"، تكون الرياضيات النظرية الشائعة تافهة. بطلب الحاجة إلى هرميات مفاهيمية عميقة، واجهنا الحاسوب الآلي بتحدي فكري جديد جذرياً والذي ليس له سابقة في التاريخ." أصبحت البرمجيات بالطبع أعقد حتى من ذلك، منذ 1989، ونسبة ديكسترا 1 إلى 109 يمكن أن تكون ببساطة أقرب إلى 1 إلى 1015 اليوم.

أشار ديكسترا أنه لا يوجد دماغ كبير كفاية ليستوعب برنامج حاسوبي حديث[2] (ديكسترا 1972)، والذي يعني، بكوننا مطوري برمجيات، لا ينبغي أن نحاول أن نحشو برامج كاملة إلى أدمغتنا دفعة واحدة؛ بل ينبغي أن

[1] يوجد طريقتين لبناء تصميم برمجي: الأولى اجعله بسيطاً جداً، عندها بشكل واضح لا يوجد أي قصور. والأخرى أن تجعله مقعداً جداً بحيث لا يوحد أي قصور واضح. -سي. ايه. آر هوار. C. A. R. Hoare

[2] أحد العلامات الدالة على أنك غصت في تعقيد مفرط هي عندما تجد نفسك مصراً على تطبيق طريقة لا ترتبط بالموضوع بشكل واضح، على الأقل لأي مراقب خارجي. هذه الحالة مثل حالة شخص مغفل بالميكانيك تعطلت سيارته-لذا ملأ البطارية بالماء وأفرغ منفضة السجائر. -ابي. جي. بلاوغير P. J. P l a u g e r

نحاول أن ننظم برامجنا بحيث نستطيع بأمان أن نركز على جزء واحد في كل مرّة. الهدف هو تقليل المقدار من البرنامج الذي يجب أن نفكر به في وقت واحد. ربما ستفكر أن هذا شعوذة فكريّة-بزيادة عدد الكرات الفكريّة التي يطلب منك البرنامج أن تكون بالهواء معاً في لحظة واحدة، يزداد احتمال أن تسقط واحدة من هذه الكرات منك، ومما يؤدي إلى مشاكل في التصميم أو كتابة الشفرة.

في مستوى هيكلة البرمجية، يُخفَّف تعقيد المشكلة بتقسيم النظام إلى أنظمة فرعيّة. يقضي البشر وقتاً أسهل باستيعاب عدّة قطع بسيطة من المعلومات من استيعاب قطعة واحدة معقّدة. الهدف من كل تقنيات تصميم البرمجيات أن تكسر المشكلة المعقدة إلى أجزاء بسيطة. كلما كانت الأنظمة الفرعية أكثر استقلالية، كلما كان بإمكانك أن تركّز بأمان على كسرة صغيرة من التعقيد في كل مرّة. الأغراض المُعرّفة بعناية تفصل بين المفاهيم بحيث يمكنك أن تركز على شيء واحد في وقت واحد. الحِزم تقدم نفس الفائدة في مستوى أعلى من التجميع. يساعد الحفاظ على الاجرائيات قصيرة بتخفيف الحمل الذهني. وكتابة البرامج بالنظر إلى زاوية المشكلة، بدلاً من النظر إلى تفاصيل التحقيق منخفضة المستوى، والعمل بأعلى مستوى من التجريد يخفف الحمل على الدماغ. الخلاصة هو أن المبرمجين الذين يحاولون تخفيف آثار محدودية الانسان المتأصلة يكتبون شفرات فهمها أسهل لهم ولغيرهم وتحتوي أخطاء أقل.

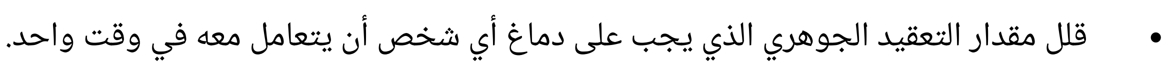
كيف تهاجم التعقيد

التصميم غير الفعّال والمكلف بشكل زائد ينشأ في ثلاث حالات:

- حل معقد لمشكلة بسيطة
- حل بسيط غير صحيح لمشكلة معقدة
- حل معقد غير مناسب لمشكلة معقدة

كما أشار ديكسترا، البرمجيات الحديثة معقدة بشكل متأصّل، ولا يهم كم حاولت بجد، ستصطدم بالنهاية بمستوى ما من التعقيد والذي هو متأصل في مشكلة الواقع نفسها. وهذا يقترح نهج بحدين لإدارة التعقيد:





- قلل مقدار التعقيد الجوهري الذي يجب على دماغ أي شخص أن يتعامل معه في وقت واحد.
- احفظ التعقيد العرضي من التكاثر غير الضروري.

عندما تدرك أنّ كل الأهداف التقنية الأخرى في البرمجية هي ثانوية بالمقارنة مع التحكم بالتعقيد، فالكثير من اعتبارات التصميم تصبح واضحة.

المميزات المرغوبة في التصميم

التصميم عالي الجودة له عدّة مميزات عامّة. إذا استطعت تحقيق كل هذه الأهداف، فإن تصميمك سيكون فعلاً جيداً جداً.[1] بعض الأهداف تتناقض مع الأهداف الأخرى، ولكن هذا هو التحدّي في التصميم، خلق مجموعة جيدة من المبادلات بين الأهداف المتنافسة. بعض مميزات جودة التصميم هي أيضاً مميزات البرنامج الجيد: الموثوقية، والأداء، وهلم جرا. والبعض الآخر مميزات داخلية للتصميم.

فيما يلي قائمة بمميزات التصميم الداخلي[2]:

الحد الأدنى من التعقيد: يجب أن يكون الهدف الأساسي للتصميم هو التقليل إلى أدنى حد من التعقيد لجميع الأسباب الموصوفة للتو. تجنب صنع تصاميم "ذكية". التصاميم الذكية عادة ما تكون صعبة الفهم. بدلا من ذلك اصنع تصاميم "بسيطة" و "سهلة الفهم".

إذا كان التصميم الخاص بك لا يسمح لك أن تتجاهل بأمان معظم الأجزاء الأخرى من البرنامج عندما تكون منغمس في جزء واحد معين، فهذا يعني أن التصميم لا يقوم بعمله.

سهولة الصيانة: سهولة الصيانة تعني التصميم لمبرمج الصيانة. تخيّل باستمرار الأسئلة التي سوف يسألها مبرمج الصيانة عن الشفرة الذي تكتبها. فكّر في مبرمج الصيانة كأنّه جمهورك المتابع، ومن ثم صمّم النظام ليكون ذاتي التفسير.

الاقتران الضعيف: الاقتران الضعيف يعني التصميم بحيث يكون عدد الاتصالات بين الأجزاء المختلفة من البرنامج بأدنى حد ممكن. استخدم مبادئ التجريد الجيّد في واجهات الصفوف، والتغليف، وإخفاء المعلومات لتصميم صفوف مع أقل عدد ممكن من الترابط. الحد الأدنى من الترابط يقلّل من العمل أثناء التكامل والاختبار والصيانة.

قابلية التوسيع: قابلية التوسيع تعني أنه يمكنك تحسين النظام دون التسبب في تخريب البنية الأساسيّة. يمكنك تغيير جزء من النظام دون التأثير على الأجزاء الأخرى. التغييرات المحتمل حدوثها أكثر تسبب للنظام أقل أذى.

---

[1] عندما أعمل على حل مشكلة لا أفكر أبدا بالجمالية. أفكر فقط بكيفية حل المشكلة. ولكن عندما أنتهي، إذا لم يكن الحل جميلاً، عندها أعلم أنه خاطئ ـــــر. باكمينستر فولر

[2] **إشارة مرجعية** ترتبط هذه المميزات بالسمات العامة لجودة البرمجيات. للحصول على تفاصيل حول السمات العامة، راجع القسم 20-1، "مميزات جودة البرمجيات".

إعادة الاستخدام: تعني إعادة الاستخدام تصميم النظام بحيث يمكنك إعادة استخدام أجزاء منه في أنظمة أخرى.

تعدد المداخل العالي (مروحة الدخل العالية): يشير مصطلح تعدد المداخل إلى وجود عدد كبير من الصفوف التي تستخدم صف معين. تعدد المداخل يدل على أنّ النظام قد تم تصميمه من أجل الاستفادة الجيّدة من صفوف الأدوات الخدمية في المستويات الدنيا في النظام.

تعدد المخارج (مروحة الخرج المنخفضة_إلى_المتوسطة): ويعني مصطلح المروحة المنخفضة إلى المتوسطة وجود صف معين يستخدم عدد منخفض إلى متوسط من الصفوف الأخرى. تشير المروحة العالية (أكثر من سبعة تقريباً) إلى أنّ الصف يستخدم عدداً كبيراً من الصفوف الأخرى، وبالتالي قد تكون معقدة للغاية. وقد وجد الباحثون أنّ مبدأ مروحة الخرج المنخفضة مفيداً إذا ما كنت تفكّر في عدد من الإجرائيات التي تُستدعى من داخل إجرائية أو عدد من الصفوف المستخدمة داخل صف (البطاقة والزجاج 1990؛ باسيلي، برياند، وميلو 1996) Card and Glass 1990; Basili, Briand, and Melo1996.

المحمولية : المحمولية تعني تصميم النظام بحيث يمكنك بسهولة نقله إلى بيئة أخرى.

الرشاقة : الرشاقة تعني تصميم النظام بحيث لا يوجد فيه أية أجزاء إضافية (ويرث 1995، مكونيل 1997). وقال فولتير أنه لا يتم الانتهاء من كتاب عندما لا شيء أكثر يمكن أن يُضاف اليه ولكن عندما لا شيء أكثر يمكن أن يُحذف منه. في البرمجيات، يعتبر هذا صحيح بشكل خاص لأنه يجب تطوير شفرة إضافيّة، ومراجعتها واختبارها، وأخذها بالاعتبار عند تعديل أجزاء أخرى من الشفرة. يجب أن تظل الإصدارات المستقبلية من البرنامج متوافقة مع الشفرة الإضافية. السؤال المصيري هو "إنّه سهل، فما هو الضرر الذي سيحدث عند إضافته؟"

الطبقية (التقسيم الطبقي): التقسيم الطبقي يعني محاولة الحفاظ على مستويات من الطبقات المتفككة بحيث يمكنك عرض النظام على أي مستوى معين ومعاينته بشكل مستقل وثابت. صمّم النظام بحيث يمكنك معاينة مستوى واحد دون الانغماس في المستويات الأخرى.

على سبيل المثال، إذا كنت تكتب نظاماً جديداً عصرياً ويجب أن يستخدم الكثير من الشفرات القديمة التي تم تصميمها بشكل سيء، فأنشئ طبقة في النظام الجديد مسؤولة عن التواصل مع الشفرة القديمة[1]. صمّم الطبقة

[1] **اشارة-مرجعية** لمزيد من المعلومات حول العمل مع الأنظمة القديمة، انظر القسم 24-5، "إعادة هيكلة الاستراتيجيات".

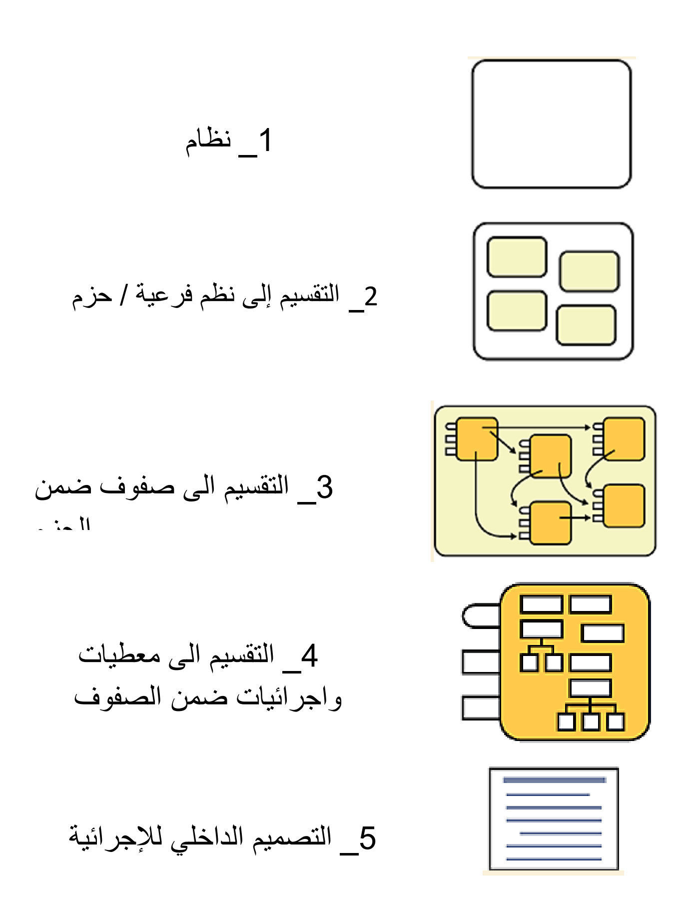
بحيث تخفي الجودة السيئة للشفرة القديمة، وتقدم مجموعة متناسقة من الخدمات إلى الطبقات الأحدث. ثم اجعل بقية النظام تستخدم تلك الصفوف بدلاً من الشفرة القديمة.

الآثار المفيدة للتصميم الطبقي في مثل هذه الحالة هي (1) أنّه يجزئ الفوضى التي تسببها الشفرة السيئة و (2) إذا سمح لك بالتخلص من الشفرة القديمة أو إعادة تصنيعها، فلن تحتاج إلى تعديل أي شفرة جديدة باستثناء طبقة الواجهة.

التقنيات المعياريّة[1]: كلما كان النظام يعتمد على الأجزاء الغريبة، كلما كان الأمر أكثر تخويفا بالنسبة لشخص يحاول فهمه للمرة الأولى. حاول أن تعطي النظام كلّه طابعاً مألوفاً باستخدام منهجيات شائعة ومعياريّة.

مستويات التصميم

هناك حاجة إلى التصميم على عدّة مستويات مختلفة من التفصيل في نظام البرمجيات. بعض تقنيات التصميم تنطبق على جميع المستويات، والبعض ينطبق على واحد أو اثنين فقط. ويوضح الشكل 5-2 المستويات.



[1] **اشارة-مرجعية** هناك نوع خاص من المعياريّة هو استخدام نماذج التصميم، التي تُناقش في "ابحث عن نماذج تصميم شائعة" في القسم 5.3.

الشكل 5-2 مستويات التصميم في برنامج. النظام (1) يقسم أولا إلى نظم فرعية (2). وتنقسم النظم الفرعية إلى صفوف (3)، وتنقسم الصفوف إلى اجرائيات ومعطيات (4). ويتم تصميم داخل كل اجرائية أيضاً (5).

المستوى 1: نظام البرمجيات

المستوى الأول هو النظام بأكمله[1]. ينتقل بعض المبرمجين مباشرة من مستوى النظام إلى تصميم الصفوف، ولكن من المفيد عادةً التفكير بمستويات أعلى من تراكيب الصفوف، مثل النظم الفرعيّة أو الحزم.

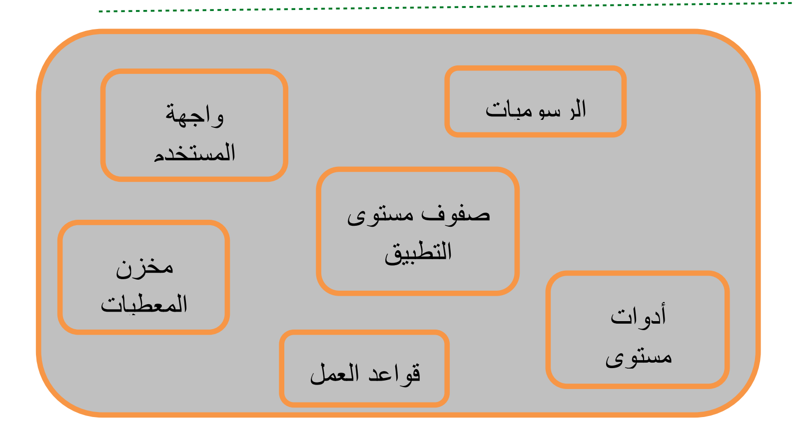
المستوى 2: التقسيم إلى نظم فرعية أو حزم

النتاج الرئيسي للتصميم في هذا المستوى هو تحديد جميع النظم الفرعية الرئيسية. النظم الفرعية يمكن أن تكون كبيرة: قاعدة البيانات، واجهة المستخدم، قواعد العمل، مترجم الأوامر، مولّد التقارير، وهلم جرا. النشاط الرئيسي للتصميم في هذا المستوى هو تحديد كيفية تقسيم البرنامج إلى النظم الفرعية الرئيسية وتحديد كيفية السماح لكل نظام فرعي باستخدام كل نظام فرعي آخر. وعادةً نحتاج التقسيم على هذا المستوى في أي مشروع، ويستغرق التقسيم وقتاً أطول من بضعة أسابيع. وضمن كل نظام فرعي، يمكن استخدام أساليب مختلفة للتصميم - اختيار النهج الذي يناسب كل جزء من أجزاء النظام. في الشكل 5-2، التصميم في هذا المستوى مرقم بـ2.

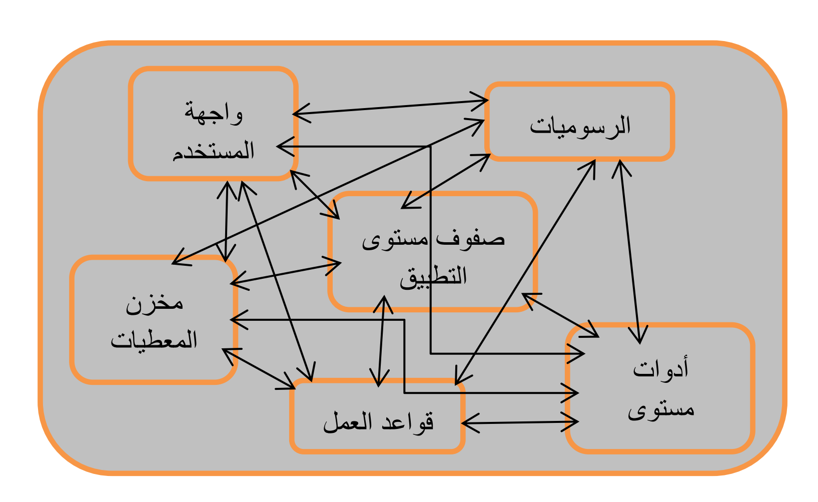
ومما له أهمية خاصّة في هذا المستوى القواعد المتعلقة بكيفيّة اتصال مختلف النظم الفرعيّة. إذا كانت جميع النظم الفرعية يمكنها التواصل مع جميع النظم الفرعية الأخرى، عندها تفقد الفائدة من فصلها أساساً. اجعل كل نظام فرعي ذو مغزى من خلال تقييد الاتصالات.

لنفترض على سبيل المثال أنّك تحدّد نظاماً من ستة أنظمة فرعيّة، كما هو مبيّن في الشكل 5-3. عندما لا توجد قواعد، سوف يتطبق القانون الثاني من الديناميكا الحرارية وسوف تزداد العشوائية في النظام. إحدى الطرق التي تزداد فيها العشوائية هو أنه بدون فرض أي قيود على الاتصالات بين الأنظمة الفرعية، سيحدث الاتصال بطريقة غير مقيدة، كما هو موضح في الشكل 5-4.

[1] بعبارة أخرى - وهذا المبدأ يعد حجر الأساس الذي تأسس عليه نجاح شركة غالاكسي وايد - عيوب التصميم الجوهرية تكون مخفية تماما من قبل عيوب التصميم الظاهرية. ___دوجلاس آدمز



الشكل 5-3 مثال لنظام من ستة أنظمة فرعية.



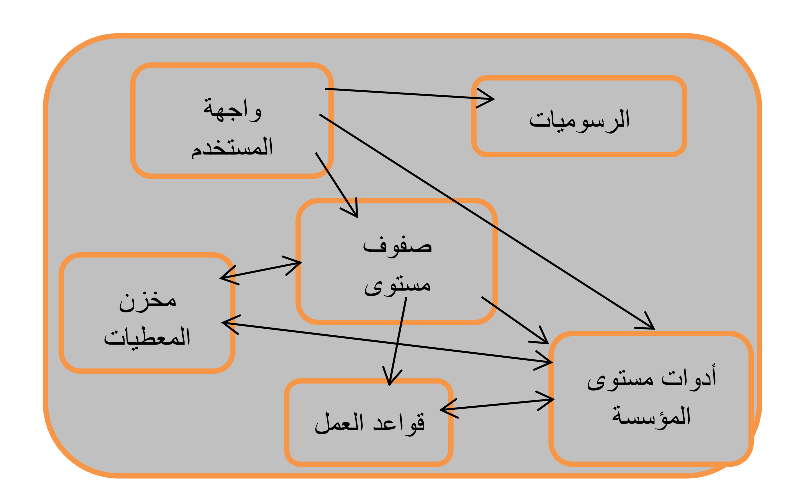
الشكل 5-4 مثال لما يحدث عند عدم وجود قيود على الاتصالات بين الأنظمة الفرعية.

كما ترون، كل نظام فرعي يتصل مباشرة مع كل نظام فرعي آخر، الأمر الذي يثير بعض الأسئلة الهامّة:

- كم من أجزاء النظام المختلفة يحتاج المطور ليفهم – على الأقل شيء بسيط – ليغير شيئا في نظام الرسوميات الفرعي؟
- ماذا يحدث عند محاولة استخدام "قواعد العمل" في نظام آخر؟
- ماذا يحدث عندما تريد وضع واجهة مستخدم جديدة على النظام، ربما واجهة سطر الأوامر لأغراض الاختبار؟
- ماذا يحدث عندما تريد وضع مخزن البيانات على جهاز بعيد؟

يمكنك أن تفكّر في الخطوط الفاصلة بين النظم الفرعية باعتبارها خراطيم يمر عبرها الماء. إذا أردت الوصول إلى نظام فرعي وسحبه، فإنّ هذا النظام الفرعي سيكون لديه بعض الخراطيم المتصلة به. كلما ازداد عدد الخراطيم التي يتوجب عليك فصلها ووصلها، كلما ازدادت كمية البلل التي ستتعرض لها. ربما ترغب في هندسة النظام الخاص بك بحيث إذا قمت بسحب نظام فرعي لاستخدامه في مكان آخر، لن يكون لديك الكثير من الخراطيم لإعادة توصيلها وهذه الخراطيم سيعاد توصيلها بسهولة.

مع امعان النظر في العواقب، كل هذه القضايا يمكن معالجتها مع القليل من العمل الإضافي. اسمح بالتواصل بين النظم الفرعية فقط على أساس "الحاجة إلى معرفة" – ويفضّل أن يكون سبباً وجيهاً. إذا وقعت في شك، فإنّه من الأسهل تقييد الاتصالات في وقت مبكر وفضفضتها لاحقاً من أن تقوم بفضفضتها في وقت مبكر ومن ثم محاولة تقييدها بعد أن تكون قد قمت بترميز عدة مئات من الاستدعاءات بين الأنظمة الفرعية. ويبين الشكل 5-5 كيف يمكن لبعض الاتصالات الموجزة أن تغيّر النظام المبيّن في الشكل 5-4.



الشكل 5-5 مع بعض قواعد الاتصال، يمكنك تبسيط التفاعلات النظام الفرعي بشكل كبير.

للحفاظ على الاتصالات سهلة الفهم والصيانة، والخطأ في جهة العلاقات بين الأنظمة الفرعية بسيط. أبسط علاقة هو أن يكون النظام الفرعي يستدعي اجرائية في نظام فرعي آخر. وهناك علاقة أكثر ارتباطا هو أن يكون نظام فرعي يحتوي على صفوف من نظام فرعي آخر. العلاقة الأكثر ارتباطاً هو أن تكون صفوف في نظام فرعي ترث من صفوف في نظام فرعي آخر.

قاعدة عامّة جيّدة هي أن مخطط لمستوى النظام مثل الشكل 5-5 يجب أن يكون مخطط غير حلقي. بعبارة أخرى، لا يجب أن يحتوي برنامج على علاقة حلقية بحيث يكون فيها: الصف أ يستخدم الصف ب، الصف ب يستخدم الصف ج، الصف ج يستخدم الصف أ.

في البرامج الضخمة وعائلات البرامج، التصميم على مستوى النظم الفرعية يحدث فرقاً. إذا كنت تعتقد أنّ البرنامج الخاص بك صغير بما يكفي لتخطي تصميم مستوى النظم الفرعية، فعلى الأقل اجعل قرارك بتخطي هذا المستوى من التصميم قرارا واعياً.

الأنظمة الفرعية الشائعة: بعض أنواع الأنظمة الفرعية تظهر مراراً وتكراراً في مختلف الأنظمة. وهذه قائمة ببعض تلك الأنواع المعتادة:

قواعد العمل[1]: قواعد العمل هي القوانين، والضوابط، والسياسات، والاجراءات التي تبرمجها في نظام حاسوبي. إذا كنت تكتب نظام رواتب، يمكنك أن تقوم ببرمجة قواعد من هيئة الضرائب حول عدد الاستقطاع المسموح به ومعدل الضريبة المقدر. قد تأتي قواعد إضافية لنظام الرواتب من عقد نقابي يحدّد معدلات العمل الإضافي، وأجور الإجازات والعطلات، وهلم جرا. إذا كنت تكتب برنامجا لتحديد أسعار التأمين على السيارات، فقد تأتي القواعد من اللوائح الحكومية المتعلقة بتغطية المسؤولية المطلوبة، أو جداول تخمين الأسعار، أو قيود التأمين.

واجهة المستخدم: أنشئ نظاماً فرعياً لعزل مكونات واجهة المستخدم بحيث يمكن تطوير واجهة المستخدم دون الإضرار بباقي البرنامج. في معظم الحالات، يستخدم النظام الفرعي لواجهة المستخدم العديد من الأنظمة الفرعية أو الصفوف من أجل واجهة المستخدم الرسومية، واجهة سطر الأوامر، عمليات القائمة، إدارة النوافذ، نظام المساعدة، وهكذا دواليك.

الوصول لقاعدة المعطيات: يمكنك إخفاء تفاصيل التنفيذ المتعلقة بالوصول إلى قاعدة المعطيات بحيث أنه لا داعي بأن يهتم معظم البرنامج بالتفاصيل الفوضوية الخاصّة بالتعامل مع بنى المعطيات منخفضة المستوى، ويمكن التعامل مع البيانات من حيث كيفية استخدامها على مستوى مشاكل الأعمال. وتوفّر النظم الفرعية التي تخفي تفاصيل التنفيذ مستوى ثمين من التجريد الذي يقلل من تعقيد البرنامج. وإنّها تمركز عمليات قاعدة البيانات في مكان واحد وتقلل فرصة حدوث أخطاء في العمل مع البيانات. وهي تسهل عملية تغيير هيكل تصميم قاعدة البيانات دون تغيير معظم البرنامج.

تبعيات النظام: قم بحزم تبعيات نظام التشغيل في نظام فرعي لنفس السبب الذي تقوم لأجله بحزم تبعيات الأجهزة الصلبة. إذا كنت تقوم بتطوير برنامج لـ مايكروسوفت ويندوز، على سبيل المثال، لماذا تحدّ نفسك في بيئة ويندوز فقط؟ اعزل استدعاءات ويندوز في نظام واجهة-ويندوز الفرعي. إذا رغبت لاحقاً أن تنقل البرنامج الخاص بك إلى نظام تشغيل ماكنتوش أو لينكس، كل ما عليك تغييره هو النظام الفرعي للواجهة. يمكن أن يكون النظام الفرعي للواجهة ضخماً جداً لتنفيذه بنفسك، ولكن هذه الأنظمة الفرعية متاحة بسهولة في أي من مكتبات الشيفرات التجارية المتعددة.

المستوى 3: التقسيم إلى صفوف[2]

[1] **اشارة-مرجعية** لمزيد من المعلومات حول تبسيط منطق الأعمال من خلال التعبير عن ذلك في الجداول، راجع الفصل 18 "الطرق المستندة إلى الجدول".

[2] **اقرأ أيضاً** للحصول على مناقشة جيدة لتصميم قواعد البيانات، انظر تقنيات قواعد البيانات الرشيقة (أمبلر 2003). Agile Database Techniques (Ambler 2003).

ويشمل التصميم على هذا المستوى تحديد جميع الصفوف في النظام. فعلى سبيل المثال، يمكن تقسيم نظام واجهة قاعدة البيانات الفرعي إلى صفوف النفاذ إلى البيانات وصفوف إطارات العمل الثابتة وكذلك البيانات الوصفية لقاعدة البيانات. ويبيّن الشكل 5-2، المستوى 3 كيف يمكن تقسيم أحد الأنظمة الفرعية من المستوى 2 إلى صفوف، وهو ما يعني أن الأنظمة الفرعية الثلاثة الأخرى المبيّنة في المستوى 2 تنقسم أيضاً إلى صفوف.

يتم تحديد تفاصيل الطرق التي يتفاعل بها كل صف مع بقية النظام كما يتم تحديد الصفوف. على وجه الخصوص، يتم تعريف واجهة الصف. وعموماً، فإنّ نشاط التصميم الرئيسي على هذا المستوى هو التأكّد من أن جميع النظم الفرعية قد تم تحليلها إلى مستوى من التفصيل جيد بما فيه الكفاية بحيث يمكنك تنفيذ أجزاءها كصفوف مستقلة.

وعادةً ما تكون هناك حاجة إلى تقسيم النظم الفرعية إلى صفوف في أي مشروع يستغرق وقتا أطول من بضعة أيام[1]. إذا كان المشروع كبيراً، يكون التقسيم متميزاً بشكل واضح عن تقسيم البرنامج في المستوى 2. إذا كان المشروع صغيراً جداً، يمكنك أن تنتقل مباشرة من عرض النظام بأكمله في المستوى 1 إلى عرض الصفوف في المستوى 3.

الصفوف مقابل الكائنات: مفهوم أساسي في التصميم غرضي التوجه هو التمييز بين الكائنات والصفوف. الكائن هو أي كيان محدّد موجود في البرنامج أثناء التشغيل. الصف هو الشيء الثابت الذي تنظر إليه في جدولة البرنامج. الكائن هو الشيء الديناميكي مع القيم والواصفات المحددة التي تراها عند تشغيل البرنامج. على سبيل المثال، يمكنك تعريف الصف "شخص" له واصفات الاسم، العمر، الجنس وما إلى ذلك. في أثناء التشغيل سيكون لديك الكائنات نانسي، هانك، ديان، توني، وهلم جرا، وهذه هي، حالات محددة من الصف. إذا كان مصطلح قاعدة البيانات مألوفا بالنسبة لك، فهو نفس التمييز بين "المخطط" و "النموذج". يمكنك تشبيه الصف بقاطع الكعكة وتشبيه الكائن بالكعكة. يستخدم هذا الكتاب المصطلحات بشكل غير رسمي ويشير عموماً إلى الصفوف والكائنات أكثر أو أقل بالتبادل.

المستوى 4: التقسيم إلى اجرائيات

التصميم في هذا المستوى يتضمن تقسيم كل صف إلى اجرائيات. ستحدّد واجهة الصف المعرّفة في المستوى 3 بعض من الإجرائيات. وسيقوم التصميم في المستوى 4 بتفصيل الإجرائيات الخاصّة بالصف. عندما تقوم بفحص تفاصيل الإجرائيات داخل صف، يمكنك أن ترى أن العديد من الإجراءات هي صناديق بسيطة ولكن القليل منها تتكون من إجرائيات منظمة هرمياً، والتي تتطلب المزيد من التصميم.

[1] **اشارة-مرجعية** من أجل تفاصيل أكثر حول مميزات الصفوف عالية الجودة، انظر الفصل 6."صفوف ناجحة".

وغالباً ما يؤدي إجراء تعريف كامل لإجرائيات الصف إلى فهم أفضل لواجهة الصف، ويؤدي إلى تغييرات مقابلة في الواجهة، أي التغييرات مرّة أخرى في المستوى 3.

وغالباً ما يترك هذا المستوى من التحليل والتصميم إلى المبرمج بشكل شخصي، ويكون بحاجة له في أي مشروع يستغرق أكثر من بضع ساعات. لا يلزم القيام به رسمياً، ولكن على الأقل يجب أن يتم ذهنياً.

المستوى 5: التصميم الداخلي للإجرائية[1]

التصميم على مستوى الإجرائية يتكون من وضع وظائف مفصّلة لكل من الإجرائيات بشكل فردي. وعادةً ما يترك التصميم الداخلي للإجرائية للمبرمج بشكل شخصي الذي يعمل على إجرائية مفردة. ويتكون التصميم من أنشطة مثل كتابة الشفرة الزائفة، والبحث عن خوارزميات في الكتب المرجعية، وتحديد كيفية تنظيم فقرات التعليمات البرمجية في إجرائية، وكتابة شفرة لغة البرمجة. يتم عمل هذا المستوى من التصميم دائما، على الرغم من أنّه في بعض الأحيان يتم ذلك دون وعي وبشكل سيء بدلاً من القيام به بشكل واعي وجّيد. في الشكل 5-2، التصميم في هذا المستوى مرقم بـ5.

3-5 أحجار بناء التصميم: الاستدلالات

مطوري البرمجيات يحبون أن تكون إجاباتنا مقطّعة ومجفّفة: "افعل أ وب وج، وسوف تتبعها ض وظ وغ في كل مرة". ونحن نفتخر في تعلم مجموعات غامضة من الخطوات التي تنتج الآثار المرجوة، ونغضب عندما لا تعمل التعليمات على النحو المعلن عنها. هذه الرغبة في السلوك الحتمي مناسبة للغاية لبرمجة الحاسوب المفصّلة، حيث أنّ هذا النوع من الاهتمام الدقيق بالتفاصيل يصنع أو يكسر البرنامج. ولكن تصميم البرمجيات هو قصّة تختلف عن ذلك كثيراً.

لأنّ التصميم غير حتمي، فإنّ تطبيق ماهر لمجموعة فعّالة من الاستدلالات هو النشاط الأساسي في تصميم برمجيات جيّدة. تصف الأقسام الفرعية التالية عدداً من الاستدلالات - طرق للتفكير في التصميم الذي ينتج في بعض الاوقات رؤى التصميم الجيد. يمكنك تشبيه الاستدلال كدليل للمحاكمات في "التجربة والخطأ". كنت بلا شك قد صادفت بعض من هذه من قبل. وبالتالي، تصف الأقسام الفرعية التالية كل من الاستدلالات من حيث الضرورة التقنية الأساسية للبرامج: إدارة التعقيد.

[1] **اشارة-مرجعية:** للحصول على تفاصيل حول إنشاء إجرائيات عالية الجودة، انظر الفصل 7، "اجرائيات عالية الجودة"، والفصل 8 "البرمجة الوقائية".

ايجاد كائنات العالم الحقيقي [1]

المنهج الأول والأكثر شعبية لتحديد بدائل التصميم هو "من خلال الكتاب" المنهج غرضي التوجه، الذي يركز على تحديد كائنات العالم الحقيقي والاصطناعي.

خطوات التصميم مع الكائنات هي:

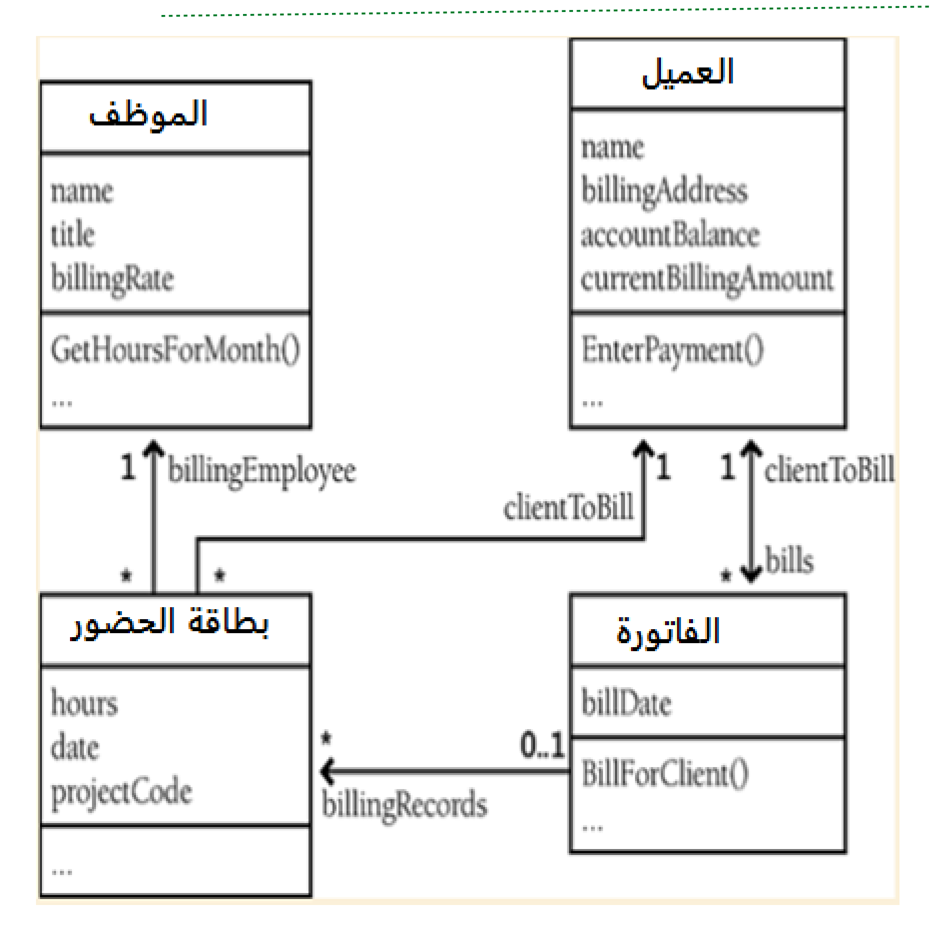
- تحديد الكائنات وخصائصها (الطرائق والمعطيات)[2].
- تحديد ما يمكن القيام به لكل كائن.
- تحديد ما يسمح لكل كائن القيام به إلى كائنات أخرى.
- تحديد الأجزاء في كل كائن التي من شأنها أن تكون مرئية لكائنات أخرى – الأجزاء التي ستكون عامّة والأجزاء التي ستكون خاصّة.
- تحديد الواجهة العامة لكل كائن.

هذه الخطوات لا يتم بالضرورة تنفيذها بالترتيب، وغالبا ما يتم تكرارها. التكرار مهم. ويرد أدناه موجز لكل من هذه الخطوات.

تحديد الكائنات وخصائصها: تعتمد برامج الحاسوب عادةً على كيانات في العالم الحقيقي. على سبيل المثال، يمكنك إنشاء نظام الفوترة الزمنية على الموظفين، العملاء، البطاقات الزمنية، والفواتير في العالم الحقيقي. ويبين الشكل 5-6 عرض غرضي التوجه لنظام الفوترة هذا.

[1] لا تسأل أولا ماذا يفعل النظام؛ اسأل ما هو الشيء الذي يُفعل ذلك له!_ بيرتراند ماير

[2] **إشارة مرجعية** لمزيد من التفاصيل حول التصميم باستخدام الصفوف، راجع الفصل 6 "صفوف ناجحة".



الشكل 5-6 يتكون نظام الفوترة هذا من أربعة كائنات رئيسية. تم تبسيط الكائنات لهذا المثال.

إنّ تحديد واصفات الكائنات ليس أكثر تعقيدا من تحديد الكائنات نفسها. فكل كائن له خصائص ذات صلة ببرنامج الحاسوب. على سبيل المثال، في نظام الفوترة الزمنية، يكون لكائن الموظف اسم، وعنوان، ومعدل تحرير الفواتير. يحتوي كائن العميل على اسم، وعنوان إرسال الفواتير ورصيد الحساب. يحتوي كائن الفاتورة على مبلغ الفاتورة، واسم العميل، وتاريخ الفاتورة وما إلى ذلك.

تتضمن الكائنات في نظام واجهة المستخدم الرسومية النوافذ، ومربعات الحوار، والأزرار، والخطوط وأدوات الرسم. وقد يؤدي إجراء مزيد من الدراسة لنطاق المشكلة إلى خيارات أفضل للكائنات البرمجية أكثر من مطابقة واحد-لواحد مع كائنات العالم الحقيقي، ولكن كائنات العالم الحقيقي تعدّ مكان جيد للبدء منه.

تحديد ما يمكن القيام به لكل كائن: يوجد مجموعة متنوعة من العمليات يمكن القيام بها على كل كائن. في نظام الفوترة الموضح في الشكل 5-6، يمكن لكائن الموظف تغيير في العنوان أو معدل الفوترة، أو يمكن لكائن العميل تغيير اسمه أو تغيير عنوان إرسال الفواتير، وهكذا.

تحديد ما يسمح أن يفعله كل كائن مع الكائنات الأخرى: هذه الخطوة هي تماماً كما ما تبدو عليه. الأشياء العامّة التي يمكن للكائنات القيام بها لبعضها البعض هي الاحتواء والوراثة. ما هي الكائنات التي يمكن أن تحتوي على أي من الكائنات الأخرى؟ ما هي الكائنات التي يمكن أن ترث من أي من الكائنات الأخرى؟ في الشكل 5-6،

يمكن أن يحتوي كائن البطاقة الزمنية على كائن الموظف وكائن العميل، ويمكن أن يحتوي كائن الفاتورة على واحدة أو أكثر من البطاقات الزمنية. بالإضافة إلى ذلك، يمكن أن تشير الفاتورة إلى أنه تم إصدار فواتير للعميل، ويمكن للعميل إدخال دفعات لفاتورة. ومن شأن نظام أكثر تعقيداً أن يتضمن تفاعلات إضافية.

تحديد الأجزاء المرئية من قبل الكائنات الأخرى لكل كائن[1]: واحدة من قرارات التصميم الرئيسية هي تحديد أجزاء من الكائن الذي ينبغي أن تكون عامّة وتلك التي ينبغي أن تبقى خاصّة. ويجب اتخاذ هذا القرار لكل من المعطيات والطرائق.

تعريف واجهات كل كائن: تعريف الواجهات الرسميّة والنحويّة و"مستوى اللغة البرمجيّة" لكل كائن. المعطيات والطرائق التي يعرضها الكائن لكل كائن آخر تدعى "الواجهة العامّة" للكائن. وأجزاء الكائن التي يعرضها للكائنات المشتقة منه عبر الوراثة تدعى "الواجهة المحميّة" للكائن. فكّر في كلا النوعين من الواجهات.

عند الانتهاء من الخوض في خطوات تحقيق أعلى مستوى من تنظيم نظام غرضي التوجه، سوف تقوم بتكرار خطوتين. عليك تكرار تنظيم النظام بأعلى مستوى للحصول على تنظيم أفضل للصفوف. وسوف تتكرر أيضاً على كل من الصفوف التي حدّدتها، وتقود تصميم كل صف إلى مستوى أكثر من التفصيل.

تشكيل تجريدات ملائمة

التجريد هو القدرة على التعامل مع مفهوم بينما تتجاهل بعض تفاصيله بأمان – التعامل مع تفاصيل مختلفة على مختلف المستويات. في أي وقت تتعامل فيه مع مجموعة، فأنت تتعامل مع تجريد. إذا كنت تشير إلى كائن باعتباره "المنزل" بدلاً من مزيج من الزجاج، والخشب والمسامير، فأنت تقوم بالتجريد. إذا كنت تشير إلى مجموعة من المنازل باعتبارها "بلدة"، فأنت تقوم بعمل تجريد آخر.

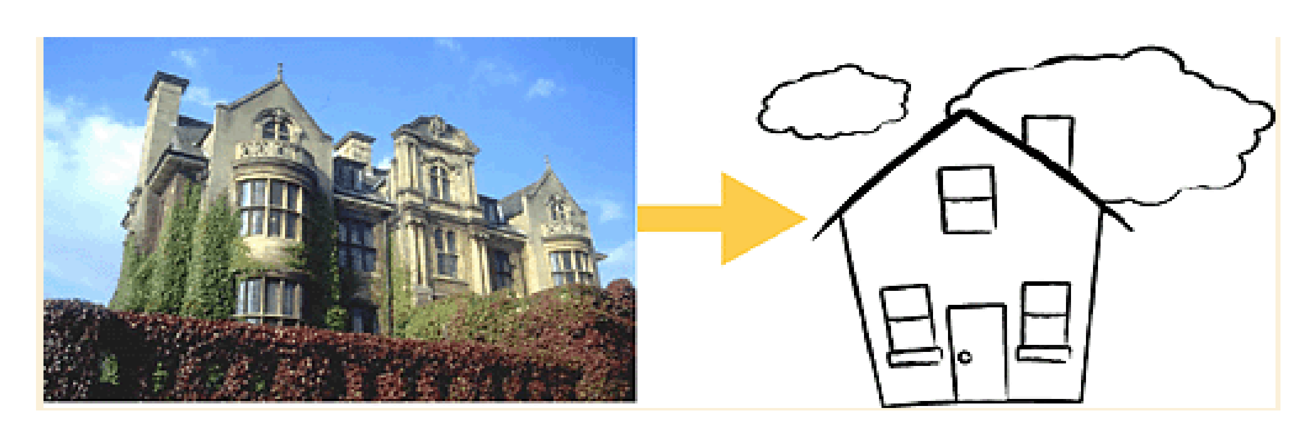
الصفوف القواعد هي التجريدات التي تسمح لك بالتركيز على الواصفات المشتركة لمجموعة من الصفوف المشتقة وتجاهل تفاصيل صفوف معينة أثناء العمل على صف القاعدة. الواجهة الجيدة للصف هي التجريد الذي يسمح لك بالتركيز على الواجهة دون الحاجة إلى القلق بشأن العمل الداخلي للصف. الواجهة لإجرائية مصمّمة تصميماً جيداً يوفر نفس الفائدة عند مستوى أدنى من التفصيل، والواجهة لحزمة أو لنظام فرعي مصمم بشكل جيد يوفر هذه الفائدة إلى مستوى أعلى من التفصيل.

من وجهة نظر معقدة، الفائدة الرئيسيّة من التجريد هي أنّه يسمح لك بتجاهل تفاصيل ليست ذات صلة. معظم كائنات العالم الحقيقي هي بالفعل تجريدات من نوع ما. كما ذكر للتو، المنزل هو تجريد للنوافذ، والأبواب،

[1] **اشارة-مرجعية:** للحصول على تفاصيل حول الصفوف واخفاء المعلومات، راجع "إخفاء الأسرار (إخفاء المعلومات)" في القسم 5. 3.

والكساء الخارجي، والأسلاك، والسباكة، والعزل وطريقة معينة لتنظيمها. الباب هو بدوره تجريد لترتيب معين من قطعة مستطيلة من مادة ما مع مفصّلات ومقبض الباب. ومقبض الباب أيضاً هو تجريد لتشكيل معين من النحاس، أو النيكل، أو الحديد، أو الفولاذ.

يستخدم النّاس التجريد بشكل مستمر. إذا كنت تتعامل مع الألواح الخشبية، وجزيئات الورنيش، وجزيئات الفولاذ بشكل مفرد في كل مرّة تستخدم الباب الأمامي الخاص بك، سيكون من الصعب عليك الدخول أو الخروج من منزلك كل يوم. كما يوحي الشكل 5-7، التجريد هو جزء كبير من كيفية التعامل مع التعقيد في العالم الحقيقي.



الشكل 5-7 التجريد يسمح لك أن تأخذ نظرة أبسط عن مفهوم معقد.

أحيانا يقوم مطورو البرمجيات ببناء نظم على مستوى اللوح الخشبي، وجزيء الورنيش وجزيء الفولاذ[1]. وهذا يجعل الأنظمة معقّدة للغاية ومن الصعب إدارتها فكريّاً. فعندما يفشل المبرمجون في توفير تجريدات برمجية كبيرة، فالنظام نفسه في بعض الأحيان يفشل في الخروج من الباب الأمامي.

يُنشئ المبرمجون الجيدون التجريدات على مستوى واجهة الإجرائية، ومستوى واجهة الصف ومستوى واجهة الحزمة – بمعنى آخر، مستوى مقبض الباب، ومستوى الباب ومستوى المنزل - وهذا يدعم برمجة أسرع وأكثر أماناً.

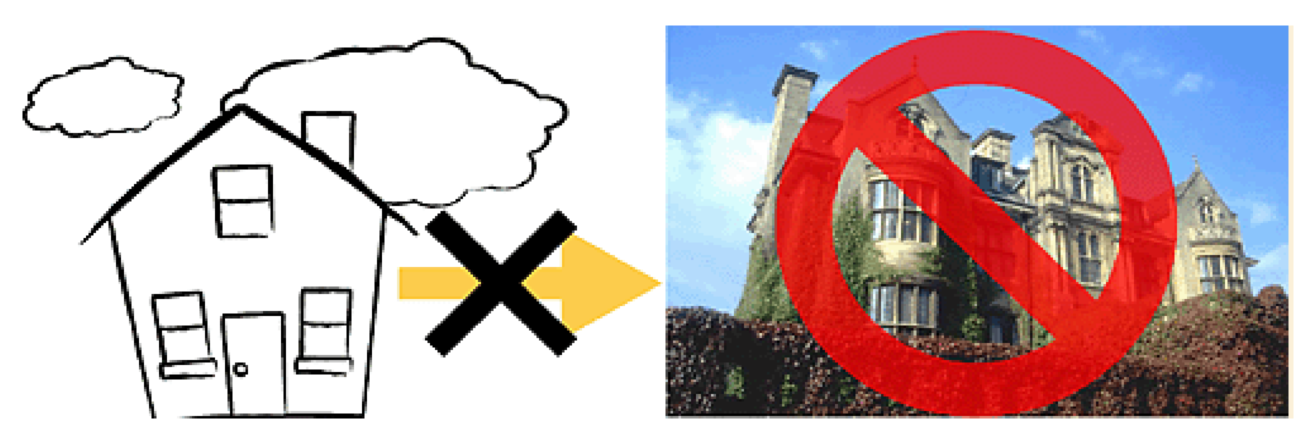
تغليف تفاصيل التنفيذ

التغليف يبدأ في المكان الذي ينتهي عنده التجريد. يقول التجريد: "يسمح لك بالنظر إلى كائن من مستوى عالٍ للتفاصيل". ويقول التغليف، "وعلاوةً على ذلك، لا يسمح لك أن تنظر إلى كائن من أي مستوى آخر للتفاصيل." بالاستمرار في قياس مواد المنزل: التغليف هو وسيلة للقول بأنه يمكنك أن تنظر للمنزل من الخارج ولكن لا

[1] **اشارة-مرجعية:** لمزيد من التفاصيل حول التجريد في تصميم الصف، انظر "التجريد الجيد" في القسم 6-2.

يمكنك الاقتراب بما فيه الكفاية لصنع تفاصيل الباب. ويسمح لك أن تعرف أن هناك الباب، ويسمح لك أن تعرف ما إذا كان الباب مفتوحاً أو مغلقاً، ولكن لا يسمح لك أن تعرف ما إذا كان الباب مصنوع من الخشب، الزجاج، الفولاذ أو بعض المواد الأخرى، وبالتأكيد لا يسمح لك بالنظر إلى كل لوح خشبي بشكل مفرد.

وكما يشير الشكل 5-8، فإن التغليف يساعد على إدارة التعقيد عن طريق منعك من النظر إلى التعقيد. يقدم القسم المعنون "التغليف الجيد" في القسم 6-2 المزيد من المعلومات الأساسية عن التغليف من حيث تطبيقه على تصميم الصف.



الشكل 5-8 يقول التغليف أنه، لا يسمح لك فقط بإلقاء نظرة أبسط على مفهوم معقد، بل لا يسمح لك بالنظر إلى أي من تفاصيل المفاهيم المعقدة. ما تراه هو ما تحصل عليه، هو كل ما تحصل عليه!

الوراثة – عندما تبسط الوراثة التصميم

في تصميم نظام البرمجيات، سوف تجد في كثير من الأحيان كائنات تشبه إلى حد كبير كائنات أخرى، باستثناء بعض الاختلافات. في نظام محاسبة، على سبيل المثال، قد يكون لديك موظفون بدوام كامل وبدوام جزئي. معظم البيانات المرتبطة بكلا النوعين من الموظفين هي نفسها، ولكن بعضها مختلف. في البرمجة غرضية التوجه، يمكنك تحديد نوع عام من الموظفين ثم تحديد الموظفين بدوام كامل كموظفين من النوع العام، باستثناء بعض الاختلافات، والموظفين بدوام جزئي أيضاً كموظفين من النوع العام، باستثناء بعض الاختلافات. عندما تكون العملية على الموظف لا تعتمد على نوع الموظف، يتم التعامل مع العملية كما لو كان الموظف مجرد موظف عام. عندما تعتمد العملية على ما إذا كان الموظف بدوام كامل أو بدوام جزئي، يتم التعامل مع العملية بشكل مختلف.

إن تحديد التشابهات والاختلافات بين هذه الكائنات يدعى "الوراثة" لأن أنواع الموظفين المحدّدة بدوام كامل ودوام جزئي، يرثون الخصائص من نوع الموظف العام.

فائدة الوراثة هي أنها تعمل بالتآزر مع فكرة التجريد. التجريد يتعامل مع الكائنات على مستويات مختلفة من التفصيل. تذكر الباب الذي كان عبارة عن مجموعة من أنواع معينة من الجزيئات على مستوى معين، ومجموعة

من الألواح الخشبية في المستوى التالي، وشيء يبقي اللصوص خارج منزلك في المستوى اللاحق. الخشب لديه خصائص معيّنة - على سبيل المثال، يمكنك قطعه بالمنشار أو لصقه باستخدام غراء الخشب، ولوح خشب الأرز لديه الخصائص العامّة للخشب بالإضافة لبعض الخصائص المحدّدة الخاصّة به.

الوراثة تبسط البرمجة لأنك تكتب إجرائية عامّة للتعامل مع أي شيء يعتمد على خصائص الباب العامّة ومن ثم تكتب إجرائيات محدّدة للتعامل مع عمليات محددة على أنواع محددة من الأبواب. بعض العمليات، مثل open() أو close()، قد تنطبق بغض النظر عما إذا كان الباب هو باب صلب، أو باب داخلي، أو باب خارجي، أو باب شاشة، أو باب فرنسي، أو باب زجاجي منزلق. قدرة لغة لدعم عمليات مثل open() أو close() دون معرفة -حتى وقت التنفيذ- ما هو نوع الباب الذي تتعامل معه يسمى "تعددية الأشكال". اللغات غرضيّة التوجه مثل سي++، جافا، والإصدارات الأحدث من مايكروسوفت فيجوال باسيك تدعم الوراثة وتعدديّة الأشكال.

الوراثة هي واحدة من أقوى أدوات البرمجة غرضيّة التوجه. يمكن أن توفر فوائد كبيرة عند استخدامها بشكل جيد، ويمكن أن تحدث ضرر كبير عند استخدامها بشكل ساذج. لمزيد من التفاصيل، راجع "الوراثة (علاقات "هو يكون")" في القسم 6-3.

إخفاء الأسرار (إخفاء المعلومات)

إخفاء المعلومات هو جزء من أساس كل من التصميم البنيوي والتصميم غرضي التوجه. في التصميم البنيوي، مفهوم "الصناديق السوداء" يأتي من اخفاء المعلومات. وفي التصميم غرضي التوجه، فإنه يؤدي إلى مفاهيم التغليف والنمطيّة ويرتبط مع مفهوم التجريد. إخفاء المعلومات هو واحد من الأفكار المركزيّة في تطوير البرمجيات، وهذا القسم يستكشف ذلك بعمق.

وقد ظهر الاهتمام بإخفاء المعلومات للعلن أول مرّة في ورقة نشرها ديفيد برناس في عام 1972 تسمى " المعايير المستخدمة في تحليل النظم إلى وحدات" On the Criteria of BeUsed in Decomposing Systems Into Modules." وتوصف عملية إخفاء المعلومات بفكرة "الأسرار"، قرارات التصميم والتنفيذ التي يخفيها المطوّر في مكان واحد عن بقية البرنامج.

في ذكرى الطبعة العشرين لكتاب أسطورة رجل الشهر"The Mythical Man Month،" خلص فريد بروكس إلى أنّ انتقاده لإخفاء المعلومات كان واحداً من بعض الطرق التي كانت فيها الطبعة الأولى من كتابه مخطئة. "كان بارناس على حق، وكنت مخطئا حول إخفاء المعلومات"، كما أعلن (بروكس 1995). وأفاد باري بوهم أنّ إخفاء المعلومات كان تقنيّة قويّة للقضاء على تكرار العمل، وأشار إلى أنّه كان فعالاً بشكل خاص في البيئات المتزايدة والمتغيرة بشكل كبير (بوهم 1987).

إنّ إخفاء المعلومات هو مرشد قوي بشكل خاص للإلزام التقني الرئيسي للبرمجيات، لأنه، بدءا من اسمه وعبر تفاصيله، فإنّه يؤكد على إخفاء التعقيد.

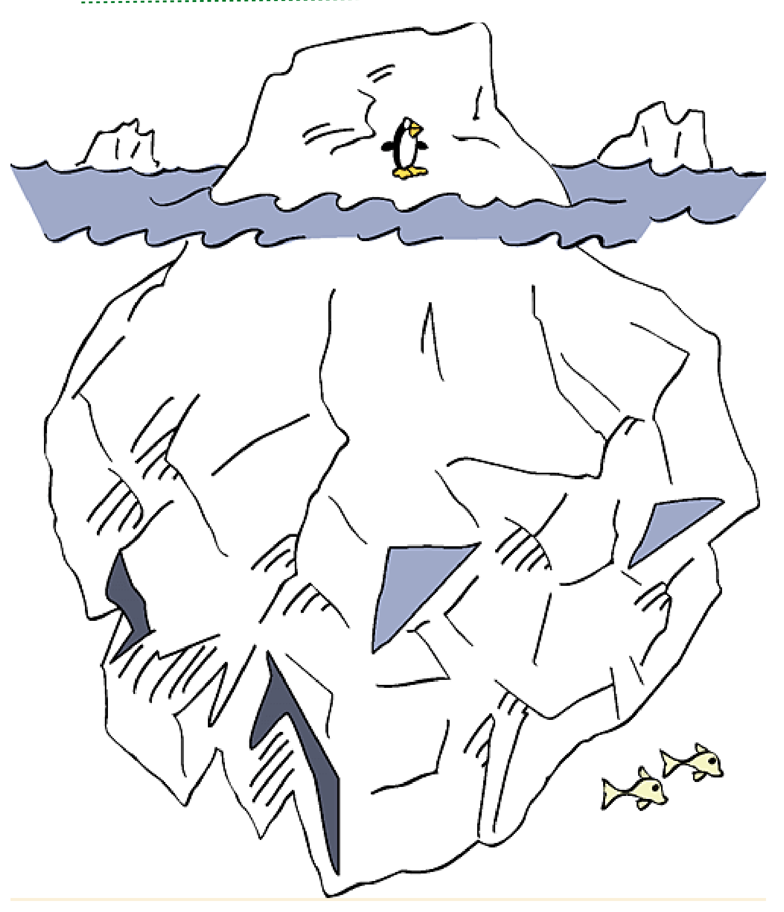
الأسرار والحق في الخصوصية

في إخفاء المعلومات، يتميز كل صف (أو حزمة أو إجرائية) بقرارات التصميم أو البناء التي يخفيها عن جميع الصفوف الأخرى. قد يكون السر منطقة من المحتمل أن تتغير أو شكل ملف أو طريقة تحقيق نوع معطيات، منطقة تحتاج إلى أن تكون معزولة عن باقي البرنامج بحيث تسبب الأخطاء في تلك المنطقة أقل قدر ممكن من الأضرار. وظيفة الصف هي الحفاظ على هذه المعلومات خفيّة وحماية حقّه في الخصوصيّة. قد تؤثر التغييرات الطفيفة على النظام على العديد من الإجرائيات داخل صف، ولكن يجب ألا يتم تمريرها خارج واجهة الصف.

إحدى المهام الرئيسية في تصميم الصف هي تحديد أي من الميزات يجب أن تُعرف خارج الصف وأي منها يجب أن تظل سريّة[1]. قد يستخدم صف 25 اجرائية ويعرض فقط 5 منهم، ويستخدم 20 أخرى داخلياً. قد يستخدم صف عدّة أنواع معطيات ولا يعرض أيّة معلومات عنها. ويعرف هذا الجانب من تصميم الصف أيضاً باسم "الرؤية" حيث يتعلق بأي من ميزات الصف تكون "مرئية" أو "معروضة" خارج الصف.

واجهة الصف يجب أن تكشف أقل قدر ممكن عن أعمالها الداخلية. كما هو مبين في الشكل 5-9، الصف يشبه الجبل الجليدي كثيرا: سبعة أثمانه تحت الماء، ويمكنك أن ترى فقط الثُمن الواحد الذي يكون فوق سطح.

[1] اسعى لتحقيق واجهات صفوف تكون كاملة وأصغرية – سكوت مايرز



الشكل 5-9 واجهة الصف الجيدة هي مثل قمة الجبل الجليدي، ويكون معظم الصف غير مكشوف.

تصميم واجهة الصف هو عملية تكرارية تماماً مثل أي جانب آخر من جوانب التصميم. إذا لم تحصل على واجهة صحيحة في المرة الأولى، حاول بضع مرات أخرى حتى يستقر. إذا لم يستقر، فأنت تحتاج لاستخدام منهج مختلف.

مثال على إخفاء المعلومات

لنفترض أنّ لديك برنامج فيه كل كائن يجب أن يملك معرف فريد (unique ID) مخزن في متغير عضو يسمى id. وتتمثل احدى منهجيات التصميم في استخدام الأعداد الصحيحة للمعرفات وتخزين أعلى معرف مخصص حتى الآن في متغير شامل (global) يسمى *g_maxId*. كلما تم تخصيص كائن جديد، ربما في الباني الخاص بكل كائن، يمكنك ببساطة استخدام العبارة *id = ++g_maxId*، والتي من شأنها أن تضمن معرف فريد، وأنه سيتم إضافة الحد الأدنى من التعليمات البرمجية في كل مرة يتم فيها إنشاء كائن. ما الخطأ الذي يمكن ان يحدث في ذلك؟

هناك الكثير من الأخطاء التي يمكن أن تحدث. ماذا لو كنت ترغب في حجز مجالات من المعرّفات لأغراض خاصّة؟ ماذا لو كنت ترغب في استخدام معرّفات غير متتابعة لتحسين الأمن؟ ماذا لو كنت تريد أن تكون قادراً على إعادة استخدام معرّفات الكائنات التي تم تدميرها؟ ماذا لو كنت ترغب في إضافة رسالة تأكيد تنطلق عند

تخصيص معرفات أكثر من الحد الأقصى الذي كنت تتوقعه؟ إذا قمت بتخصيص المعرّفات عن طريق نشر العبارة *id = ++g_maxId* في أنحاء برنامجك، سيكون عليك تغيير الشفرة المرتبطة بكل واحدة من تلك العبارات. وإذا كان برنامجك متعدد المهام، هذا المنهج لن يكون آمناً.

الطريقة التي يتم بها إنشاء المعرفات الجديدة هي قرار تصميمي يجب عليك إخفاءه. إذا كنت تستخدم العبارة *g_maxId++* طوال البرنامج، فإنك تقوم بفضح طريقة إنشاء معرف جديد، وهو ببساطة عن طريق زيادة *g_maxId*. إذا قمت بدلاً من ذلك بوضع العبارة id=NewId() في برنامجك، ستقوم بإخفاء المعلومات المتعلقة بكيفية إنشاء المعرفات الجديدة. داخل إجرائية NewId() قد لا يزال لديك سطر واحد فقط من التعليمات البرمجية، إعادة(*g_maxId++*) أو ما يعادلها، ولكن إذا قررت لاحقاً حجز مجالات معينة من المعرّفات لأغراض خاصّة أو إعادة استخدام معرفات قديمة، يمكنك اجراء تلك التغييرات داخل إجرائية NewId() نفسها – دون لمس عشرات أو مئات من عبارات id=NewId(). بغض النظر عن مدى تعقيد التعديلات التي قد تصبح داخل NewId(), فإنها لن تؤثر على أي جزء آخر من البرنامج.

الآن افترض أنك تكتشف أنك بحاجة إلى تغيير نوع المعرّف من عدد صحيح إلى سلسلة نصيّة. إذا كنت قد نشرت تصريحات المتغيرات مثل int id في أنحاء برنامجك، فإنّ استخدام إجرائية NewId() لا يساعد. لا يزال عليك الخوض خلال برنامجك والقيام بعشرات أو مئات التغييرات.

سر إضافي لإخفائه هو نوع معطيات المعرّف. من خلال كشف حقيقة أن المعرّفات هي أعداد صحيحة، فأنت تشجع المبرمجين لتنفيذ عمليات الاعداد الصحيحة عليها مثل >، <، =. في سي++، يمكنك استخدام typedef لتعريف الـ IDs الخاصّة بك لتكون من النوع IdType – المحدّد من المستخدم والذي يحل إلى int بدلاً من التعريف عنها مباشرة أنّها من نوع int. بدلاً من ذلك، في سي++ وغيرها من اللغات يمكن أن تنشىء صف IdType بسيط. مرة أخرى، اخفاء قرار التصميم يحدث فرقاً كبيراً في كميّة التعليمات البرمجية المتأثرة بالتغيير.



إنّ إخفاء المعلومات مفيد على جميع مستويات التصميم، بدءاً من استخدام الثوابت المسماة بدلاً من الحرفية، إلى انشاء أنواع المعطيات، إلى تصميم الصف، وتصميم الإجرائية وتصميم النظام الفرعي.

فئتين من الأسرار

الأسرار في إخفاء المعلومات تقع في معسكرين عامين:

- إخفاء التعقيد بحيث أن دماغك ليس عليه التعامل معه إلا إذا كنت مهتماً به على وجه التحديد.
- إخفاء مصادر التغيير بحيث تكون التأثيرات متمركزة في مكان واحد عندما يحدث التغيير.

تتضمن مصادر التعقيد أنواع المعطيات المعقّدة وهياكل الملفات والاختبارات المنطقية والخوارزميات المعنية، وما إلى ذلك. يرد لاحقاً في هذا الفصل قائمة شاملة لمصادر التغيير.

عوائق إخفاء المعلومات[1]

وفي بعض الحالات، يكون من المستحيل حقاً إخفاء المعلومات، ولكن معظم الحواجز التي تعترض إخفاء المعلومات هي حواجز عقليّة ناشئة من الاستخدام المعتاد لتقنيات أخرى.

التوزيع المفرط للمعلومات ويتمثل أحد العوائق الشائعة أمام إخفاء المعلومات في التوزيع المفرط للمعلومات في أنحاء النظام. من الممكن أن تكون قد قمت باستخدام الرقم 100 كما هو في مواضع حرجة في أنحاء النظام. استخدام 100 كما هو يلغي مركزيّة المرجعيّة له. من الأفضل إخفاء المعلومات في مكان واحد، ربما باستخدام الثابت MAX_EMPLOYEES، والذي يتم تغيير قيمته في مكان واحد فقط.

وثمّة مثال آخر على التوزيع المفرط للمعلومات هو تداخل التفاعلات مع المستخدمين في أنحاء النظام. إذا تغيّر نمط التفاعل، لنقل، من واجهة المستخدم الرسوميّة إلى واجهة سطر الأوامر، عملياً فإنّ كل الشفرة البرمجية يجب تعديلها. فمن الأفضل تركيز التفاعل مع المستخدم في صف أو حزمة أو نظام فرعي واحد بحيث يمكنك تغييره دون التأثير على النظام بأكمله.

مثال آخر هو "عنصر البيانات الشامل" (global data element) - ربما مصفوفة من بيانات الموظفين من 1000 عنصر كحد أقصى التي يتم الوصول إليها في جميع أنحاء البرنامج[2]. إذا استخدم البرنامج البيانات الشاملة مباشرة، سيتم نشر المعلومات حول تحقيق عنصر البيانات- مثل كونه مصفوفة بحد أقصى من 1000 عنصر- في جميع أنحاء البرنامج. إذا استخدم البرنامج البيانات من خلال إجرائيات الوصول فقط، فإنّ إجرائيات الوصول فقط من ستعرف تفاصيل التحقيق.

التبعيات الحلقيّة (الدائرية) وهناك حاجز أكثر خداعاً أمام إخفاء المعلومات هو التبعيات الحلقيّة، كما هو الحال عندما تستدعي إجرائية في الصف أ إجرائية في الصف ب، وتستدعي إجرائية في الصف ب إجرائية في الصف أ.

---

[1] **اقرأ أيضاً** أجزاء من هذا القسم تم تكيفها من "تصميم البرمجيات لسهولة التمدد والانكماش" (برناس 1979). ("Designing Software for Ease of Extension and Contraction" (Parnas

[2] **اشارة-مرجعية** لمزيد من المعلومات حول الوصول إلى البيانات الشاملة من خلال واجهات الصفوف، راجع "استخدام اجرائيات الوصول بدلا من البيانات الشاملة" في القسم 13-3.

تجنّب حلقات التبعية هذه. إنّها تجعل اختبار النظام صعباً لأنّه لا يمكنك اختبار الصف أ أو الصف ب حتى يكون جزء على الأقل من الآخر جاهزاً.

خطأ التشبيه بين صفوف المعطيات والبيانات الشاملة إذا كنت مبرمج تعمل بضمير، فإنّ واحد من الحواجز التي تحول دون إخفاء المعلومات بفعالية قد يكون اعتقادك أن صفوف المعطيات هي بيانات شاملة وتتجنبها لأنك تريد تجنب المشاكل المرتبطة بالبيانات الشاملة. في حين أن الطريق إلى جحيم البرمجة معبّد بالمتغيرات الشاملة، صفوف المعطيات تعرّض لمخاطر أقل بكثير.

تخضع البيانات الشاملة عموماً لمشكلتين: تعمل الإجرائية على البيانات الشاملة دون أن تعرف أن هناك إجرائيات أخرى تعمل عليها، والإجرائيات تدرك أن إجرائيات أخرى تعمل على البيانات الشاملة ولكنها لا تعرف بالضبط ما تقوم به عليها. لا تخضع صفوف البيانات لأي من هذه المشكلات. يقتصر الوصول المباشر إلى البيانات إلى عدد قليل من الإجرائيات المنظمة في صف واحد. تدرك الإجرائيات أن إجرائيات أخرى تعمل على البيانات، وهي تعرف بالضبط ما هي تلك الإجرائيات الأخرى.

وبطبيعة الحال، يفترض هذا النقاش أنّ نظامك يستفيد من الصفوف الصغيرة المصمّمة بشكل جيد. إذا كان برنامجك مصمم لاستخدام الصفوف الضخمة التي تحتوي كل منها على العشرات من الإجرائيات، فإن التمييز بين صفوف البيانات والبيانات الشاملة يصبح غير واضحاً وصفوف البيانات ستكون عرضةً لكثير من المشاكل نفسها التي تتعرض لها البيانات الشاملة.

فهم عقوبات الأداء[1] الحاجز النهائي أمام إخفاء المعلومات يمكن أن يكون محاولة تجنب عقوبات الأداء على المستويين المعماري والترميزي. لا داعي للقلق على أي مستوى. على مستوى الهيكلة، ليس هناك ضرورة للقلق لأن تصميم نظام لإخفاء المعلومات لا يتعارض مع تصميمه للأداء. إذا أبقيت على كل من إخفاء المعلومات والأداء في الاعتبار، فيمكنك تحقيق كلا الهدفين.

والقلق الأكثر شيوعاً هو على مستوى الترميز. والقلق هو أن الوصول إلى عناصر البيانات يكبّد بشكل غير مباشر عقوبات على الأداء في زمن التنفيذ لمستويات إضافية من تهيئة الكائنات، واستدعاء الاجرائيات، وما إلى ذلك. وهذا القلق سابق لأوانه. حتى تتمكن من قياس أداء النظام وتحديد الاختناقات، أفضل طريقة للتحضير لأداء العمل على مستوى الشفرة هو إنشاء تصميم وحدات عالي المستوى. عند اكتشافك النقاط الساخنة في وقت لاحق، يمكنك تحسين الصفوف والإجرائيات الفردية دون التأثير على بقية النظام.

[1] **اشارة مرجعية** للمزيد حول الوصول إلى البيانات الشاملة من خلال واجهات الصفوف، راجع "استخدام اجرائيات الوصول بدلا من البيانات الشاملة" في القسم 13. 3.

قيمة إخفاء المعلومات

بيانات مثبتة

إخفاء المعلومات هو إحدى التقنيات النظرية القليلة التي أثبتت بلا جدال قيمتها في الممارسة، والتي كانت صحيحةً لفترة طويلة (بوهم 1987a) (Boehm 1987). منذ سنوات مضت وجد أن البرامج الكبيرة التي تستخدم اخفاء المعلومات تكون أسهل في التعديل– بعامل 4 مرات– من البرامج التي لا تستخدمها (كورسون و فيشنافي 1986) (Korson and Vaishnavi 1986). وعلاوةً على ذلك، إخفاء المعلومات هو جزء من أساس كل من التصميم البنيوي والتصميم غرضي التوجه.

لإخفاء المعلومات قوّة استدلالية فريدة، وقدرة فريدة على إلهام حلول تصميمية فعّالة. يوفّر التصميم غرضي التوجه التقليدي القوة الاستدلالية لنمذجة العالم في كائنات، ولكن التفكير الغرضي لن يساعدك في تجنب الإعلان عن المعرف (ID) باعتباره عدد صحيح (int) بدلاً من IdType. " سوف يسأل المصمم غرضي التوجه، "هل ينبغي التعامل مع المعرف (ID) كأنه كائن؟" اعتمادا على معايير الترميز للمشروع، فإن الإجابة "نعم" قد تعني أنّ المبرمج عليه كتابة الباني والهادم ومعامل النسخ ومعامل الاسناد؛ والتعليق على كل شيء. ووضعه في اعدادات التحكم. سيكون قرار معظم المبرمجين، "كلا، لا يستحق الأمر إنشاء صف كامل فقط من أجل المعرف. فقط سأستخدم الأعداد الصحيحة (ints)"

لاحظ ما حدث للتو. هناك بديل تصميمي مفيد، وهو ببساطة إخفاء نوع بيانات المعرف، لم يتم حتى النظر اليه. إذا، بدلا من ذلك، لو سأل المصمم، "ماذا عن المعرّف هل يجب أن يكون مخفياً؟" لكان قد قرر إخفاء نوعه وراء التصريح عن نوع بسيط الذي يستبدل العدد الصحيح بـ IdType. والفرق بين التصميم غرضي التوجه واخفاء المعلومات في هذا المثال أكثر دقّة من صراع القواعد الصريحة والقوانين. التصميم غرضي التوجه سيوافق على قرار التصميم هذا كما ستوافق اخفاء المعلومات. بالأحرى، الفرق هو واحد من الاستدلالات – التفكير في إخفاء المعلومات يلهم ويعزز قرارات التصميم التي تفكر بها أما الأغراض لا تفعل ذلك.

إخفاء المعلومات يمكن أن يكون مفيداً أيضاً في تصميم واجهة الصف العامّة. الفجوة بين النظرية والممارسة في تصميم الصف واسعة، وبيّن العديد من مصممي الصفوف القرار حول ما يجب وضعه في واجهة الصف العامّة هو تحديد ما هي الواجهة التي ستكون أكثر ملاءمة للاستخدام، والتي عادة ما تؤدي إلى كشف أكبر قدر ممكن من الصف. مما رأيت، سيفضّل بعض المبرمجين كشف كل البيانات الخاصّة بالصف بدلاً من كتابة 10 أسطر إضافية من التعليمات البرمجية للحفاظ على أسرار الصف سليمة.

السؤال "ماذا يحتاج أن يخفي هذا الصف؟" يصل إلى قلب قضية تصميم الواجهة. إذا كان بإمكانك وضع تابع أو معطيات في واجهة الصف العامة دون كشف أسراره، قم بذلك. وغير ذلك، لا تفعل.

السؤال عمّا يجب أن يكون مخفياً يدعم قرارات التصميم الجيّد على جميع المستويات. وهو يعزز استخدام الثوابت المسمّاة بدلاً من الحرفية في مستوى البناء. وهو يساعد في خلق أسماء جيدة للإجرائيات والوسطاء داخل الصفوف. وهو يرشد القرارات المتعلقة بتحليل الصف والنظام الفرعي والاتصالات البينية في مستوى النظام.



اتخذ عادة السؤال "ماذا يجب أن أخفي؟" وسوف تتفاجأ كيف أن العديد من قضايا التصميم الصعبة سوف تذوب أمام عينيك.

تحديد المناطق المرجحة للتغيير

وجدت دراسة للمصممين العظيمين أنّ إحدى السمات التي كانت مشتركة بينهم هي قدرتهم على توقع التغيير (غلاس 1995) (Glass 1995)[1]. ويعد التكييف مع التغيرات من أكثر الجوانب صعوبة في تصميم البرامج الجيدة. والهدف من ذلك هو عزل المناطق غير المستقرة بحيث يقتصر تأثير التغيير على إجرائية أو صف أو حزمة واحدة. وفيما يلي الخطوات التي يجب اتباعها في التحضير لهذه الاضطرابات.

1. **تحديد العناصر التي من المحتمل أن تتغير.** إذا تم تنفيذ المتطلبات بشكل جيد، فإنها تتضمن قائمة بالتغييرات المحتملة واحتمالية حدوث كل تغيير منها. في مثل هذه الحالة، يكون من السهل تحديد التغيرات المحتملة. إذا كانت المتطلبات لا تغطي التغييرات المحتملة، راجع المناقشة التي تلي المناطق المحتمل تغيرها في أي مشروع.
2. **فصل العناصر التي من المحتمل أن تتغير.** قسم كل عنصر سريع التأثر تم تحديده في الخطوة 1 في صف خاص به أو في صف مع المكونات السريعة التأثر الأخرى والتي من المحتمل أن تتغير في نفس الوقت.
3. **عزل العناصر التي من المحتمل أن تتغير.** صمم واجهات الصفوف البينيّة لتكون غير حساسة للتغييرات المحتملة. صمّم الواجهات بحيث تقتصر التغييرات على ما داخل الصف ويبقى خارجه غير متأثر. أي صف أخر يستخدم الصف الذي تم تغييره يجب ألاّ يدرك بأن التغيير قد حدث. يجب على واجهة الصف أن تحمي أسرارها.

فيما يلي بعض المناطق التي من المحتمل أن تتغير:

[1] **اقرأ أيضاً** تم تكييف المنهج الموصوف في هذا القسم من "تصميم البرمجيات لسهولة التمديد والانكماش" (برناس 1979).
.Designing Software for Ease of Extension and Contraction" (Parnas 1979")

قواعد العمل[1] تميل قواعد الأعمال إلى أن تكون المصدر في التغيير المتكرر للبرامج. يغير الكونغرس الهيكل الضريبي، أو يعيد الاتحاد التفاوض بشأن عقده، أو تقوم شركة التأمين بتغيير جداول أسعارها. إذا كنت تتبع مبدأ إخفاء المعلومات، المنطق المعتمد على أساس هذه القواعد لن يكون متناثراً في أنحاء برنامجك. سيبقى المنطق مخفيّاً في زاوية مظلمة واحدة من النظام إلى أن يحتاج إلى تغيير.

تبعيات العتاد الصلب وتشمل أمثلة تبعيات العتاد الصلب واجهات للشاشات والطابعات ولوحات المفاتيح والفأرة ومحركات الأقراص ومعدات الصوت وأجهزة الاتصالات. اعزل تبعيات العتاد الصلب في نظام فرعي أو صف خاص بها. يساعدك عزل هذه التبعيات عندما تنقل البرنامج إلى بيئة عتاد صلب جديدة. كما أنه يساعدك في البداية عندما تطوّر برنامجاً لعتاد صلب سريع التأثر. يمكنك كتابة برنامج يحاكي التفاعل مع عتاد صلب معين، اجعل "النظام الفرعي لواجهة العتاد الصلب" يستخدم المحاكي طالما أنّ العتاد الصلب غير مستقر أو غير متوفر، ثم افصل "النظام الفرعي لواجهة العتاد الصلب" عن المحاكي وقم بتوصيل النظام الفرعي في العتاد الصلب عندما يكون جاهزاً للاستخدام.

الإدخال والإخراج في مستوى تصميمي أعلى قليلاً من واجهات العتاد الصلب الخام، الإدخال/الإخراج هو منطقة سريعة التأثر. إذا كان تطبيقك ينشئ ملفات البيانات الخاصّة به، فإن تنسيق الملف من المحتمل أن يتغير عندما يصبح تطبيقك أكثر تطوّراً. في مستوى المستخدم سيتم أيضاً تغيير تنسيقات الإدخال والإخراج– مواقع الحقول في الصفحة، وعدد الحقول في كل صفحة، وتسلسل الحقول– وما إلى ذلك. بشكل عام، إنّها فكرة جيدة أن تقوم بتفحص جميع الواجهات الخارجية من أجل التغييرات المحتملة.

ميزات اللغة غير القياسيّة معظم تطبيقات اللغة تحتوي على ملحقات مفيدة غير قياسية. إن استخدام الإضافات هو سيف ذو حدين لأنها قد لا تتوفر في بيئة مختلفة، سواء كانت البيئة المختلفة هي عتاد صلب مختلف أو تنفيذ مختلف للّغة من المورد أو إصدار جديد للّغة من المورد نفسه.

إذا كنت تستخدم إضافات غير قياسية للغتك البرمجية، فأخفِ تلك الإضافات في صف خاص بها بحيث يمكنك استبدالها بشفرتك الخاصة عند الانتقال إلى بيئة مختلفة. وبالمثل، إذا كنت تستخدم إجراءات مكتبة غير متوفرة في جميع البيئات، أخفِ إجراءات المكتبة الفعليّة خلف واجهة تعمل كما يجب في بيئة أخرى.

[1] **اشارة-مرجعية** واحدة من أقوى التقنيات لتوقع التغيير هي استخدام الأساليب المقادة بالجداول. لمزيد من التفاصيل، راجع الفصل 18، "أساليب مقادة بالجداول".

مناطق التصميم والبناء الصعبة إنّها فكرة جيدة أن تقوم بإخفاء مناطق التصميم والبناء الصعبة لأنّها قد تكون ضعيفة وقد تحتاج إلى القيام بها مرة أخرى. قم بتقسيمها وقلل التأثير الذي قد يسببه تصميمها أو بناؤها السيء على بقية النظام.

متغيرات الحالة متغيرات الحالة تبيّن حالة البرنامج وتتغير بشكل متكرر أكثر من معظم البيانات الأخرى. في سيناريو نموذجي، قد تقوم في الأصل بتعريف "متغير الحالة الخطأ" كمتغير من نوع منطقي وتقرر لاحقاً أنّه سيتم تطبيقه بشكل أفضل كنوع تعدادي مع القيم `ErrorType_None` و`ErrorType_Warning` و`ErrorType_Fatal`

يمكنك إضافة مستويين على الأقل من المرونة وإمكانية القراءة إلى استخدامك لمتغيرات الحالة:

- لا تستخدم متغير منطقي كمتغير للحالة. استخدام نوع تعدادي بدلاً من ذلك. من الشائع إضافة حالة جديدة إلى متغير الحالة، وإضافة نوع جديد إلى نوع تعدادي يتطلب مجرد إعادة تجميع بدلاً من مراجعة شاملة لكل سطر من الشفرة يقوم بالتحقق من المتغير.
- استخدام إجرائيات الوصول بدلاً من التحقق من المتغير مباشرة. بالتحقق من إجرائية الوصول بدلاً من المتغير، فأنت تسمح بإمكانية أكثر تطوراً للتحقق من الحالة. على سبيل المثال، إذا أردت التحقق من تركيبات تشمل متغير حالة الخطأ ومتغير حالة التابع الحالي، سيكون من السهل القيام به إذا كان الاختبار مخفياً في إجرائية ويصعب القيام به إذا كان اختبارا معقداً مشفراً في مناطق حرجة في أنحاء البرنامج.

تقييد أحجام البيانات عندما تصرح عن مصفوفة من الحجم 100، فإنك تكشف معلومات للعالم لا يحتاج العالم إلى رؤيتها. دافع عن حقك في الخصوصيّة! إن إخفاء المعلومات ليس دائما معقداً كصف كامل. في بعض الأحيان يكون الأمر بسيطاً مثل استخدام ثابت محدد مثل `Max_Employees` لإخفاء 100.

توقع درجات مختلفة من التغيير

بالتفكير في التغييرات المحتملة على النظام، قم بتصميم النظام بحيث يتناسب تأثير التغيير أو نطاقه مع احتمال حدوث التغيير[1]. إذا كان التغيير محتملاً، تأكّد من أنّ النظام يمكن أن يستوعبه بسهولة. فقط التغييرات الضعيفة الاحتمالية ينبغي أن يسمح بأن يكون لها عواقب وخيمة على أكثر من صف واحد في النظام. يشترك

[1] **اشارة-مرجعية** نهج هذا القسم لتوقع التغيير لا يتضمن التصميم مقدماً أو التشفير مقدماً. للاطلاع على مناقشة لهذه الممارسات، انظر "برنامج يحتوي على شفرة تبدو وكأنها قد تكون ضرورية في يوم من الأيام" في القسم 24-2.

أيضاً المصممون الجيدون في تقدير قيمة التغيير المتوقع. إذا كان التغيير غير محتملا بشكل كبير ولكن من السهل التخطيط له، يجب أن تفكّر بجد في التعامل معه أكثر مما لو لم يكن محتملاً جداً ويصعب التخطيط له.

هناك أسلوب جيد لتحديد المناطق التي يحتمل أن تتغير[1]، هو: أولاً تحديد المجموعة الفرعية الأصغرية من البرنامج والتي قد تكون ذات فائدة للمستخدم. تشكل المجموعة الفرعية نواة النظام ومن غير المحتمل أن تتغير. بعد ذلك، حدّد الحد الأدنى من الزيادات للنظام. يمكن أن تكون صغيرة جداً بحيث تبدو عديمة الأهمية. عند النظر في التغييرات الوظيفية، تأكّد أيضاً من النظر في التغييرات النوعية: جعل البرنامج آمن "للمسارات"، وجعله قابل للمركزية، وهلم جرا. وتفرض مجالات التحسين المحتملة هذه تغييرات محتملة في النظام؛ صمّم هذه المناطق باستخدام مبادئ إخفاء المعلومات. من خلال تحديد النواة أولاً، يمكنك معرفة المكونات التي هي حقاً إضافات ثم استقرئ وأخف التحسينات من هناك.

الحفاظ على الاقتران ضعيفاً

يصف "الاقتران" مدى شدة ارتباط صف أو إجرائية مع الصفوف والاجرائيات الأخرى. الهدف هو إنشاء صفوف واجرائيات بعلاقات صغيرة ومباشرة وواضحة ومرنة مع الصفوف والاجرائيات الأخرى، والذي يعرف ب "الاقتران الضعيف." يطبق مفهوم الاقتران بتساوٍ على الصفوف والاجرائيات، لذا حتى نهاية هذه المناقشة سأستخدم الكلمة "وحدة" لأعبر عن كلا الصفوف والاجرائيات.

يكون الاقتران الجيّد بين وحدتين ضعيفاً كفاية لدرجة أنّه من السهل استخدام إحدى الوحدتين من وحدات أخرى. تقترن عربات السكك الحديدية النموذجية بخطافات متقابلة تُقفل عندما تُضغط إلى بعضها البعض. وصل عربتين سهل-عليك فقط أن تدفعهما إلى بعضهما البعض. تخيّل كم هو الحال أصعب لو كان عليك أن تثبت الشيئين بالبراغي، أو أن تربط مجموعة من الأسلاك أو إذا كان بإمكانك أن توصل فقط أنواع محدّدة من العربات إلى أنواع محدّدة أخرى. اقتران عربات السكك الحديدية النموذجية ينجح لأنه بسيط بما فيه الإمكان. في البرمجيات، اجعل الاتصالات بين الوحدات بأبسط ما يمكن.

حاول أن تنشئ وحدات تعتمد قليلاً على الوحدات الأخرى. اجعلهم منفصلين، كما تكون علاقات العمل، بدلاً من أن يكونوا متصلين، كما يكون التوأم المتصل. إجرائية مثل `sin()` هي ضعيفة الاقتران لأن كل شيء تحتاج أن تعرفه يُمرر إليها بقيمة واحدة تمثل زاوية بالدرجات. إجرائية مثل `InitVars( var 1, var2, var3, ..., var N )` هي أشد اقتران لأن، ومع كل المتغيرات التي يجب أن تمرر، الوحدة المستدعية تعرف جزئياً ما يجري داخل `InitVars()`. الصفّين الذين يعتمد كل منهما عل استخدام الآخر لنفس البيانات الشاملة هما حتى أشد اقتراناً.

[1] **اقرأ أيضاً** وتستند هذه المناقشة إلى النهج الموصوف في "تصميم وتطوير عائلات البرامج" (برناس 1976). (On the design and development of program families" (Parnas 1976"

معايير الاقتران

هاهنا عدة معاير تستخدم في تقييم الاقتران بين الوحدات:

الحجم: يشير الحجم إلى عدد الاتصالات بين الوحدات. في الاقتران، الصغير جميل لأن الوحدات الأخرى تحتاج جهد أقل لترتبط مع وحدة تمتلك واجهة صغيرة. الإجرائية التي تأخذ وسيط واحد هي أضعف اقتران بالوحدات التي تستدعيها من إجرائية تأخذ ستة متحولات. صف بأربعة طرق عامّة معرّفة بشكل جيد أضعف اقتران بالوحدات التي تستخدمه من صف يعرض 37 طريقة عامة.

الوضوح: يشير الوضوح إلى شهرة الاتصال بين وحدتين. ليست البرمجة ككونك في المخابرات؛ لا تحصل على امتياز بكونك مستتراً. هي أشبه بالإعلانات؛ تحصل على الكثير من الامتيازات عندما تجعل اتصالاتك معروفة ما أمكن. تمرير البيانات في قائمة وسطاء يصنع اتصال واضح ولذلك هو جيد، تعديل بيانات شاملة بحيث تستطيع وحدة أخرى أن تستخدم تلك البيانات هو اتصال مستتر ولذلك هو سيء. إنّ توثيق الاتصالات بالبيانات الشاملة يجعله أكثر وضوحا وهو أفضل بقليل.

المرونة: تشير المرونة إلى سهولة تغير الاتصالات بين الوحدات، بشكل مثالي، أنت تريد شيئاً أكثر شبهاً بموصل USB في حاسوبك من الشبه بسلك معزول (عارٍ) ومسدس قصدير. المرونة جزئياً هي نتاج خصائص الاقتران الأخرى، لكنها مختلفة قليلاً أيضاً. افترض أنه لديك إجرائية تستخلص كم الاجازات التي يتلقاها الموظف كل سنة، بالاعتماد على تاريخ التوظيف وتصنيف العمل. لنسمي هذه الإجرائية ()LookupVacationBenefit. افترض أنه في وحدة أخرى لديك "كائن موظف" يتضمن تاريخ التوظيف و تصنيف العمل، بين الأشياء الأخرى، وتلك الوحدة تمرر الكائن إلى ()LookupVacationBenefit.

من وجهة نظر المعايير الأخرى، تبدو الوحدتان مقترنتان بضعف. اتصال الموظف بين الوحدتين واضح، وهناك فقط اتصال واحد. الآن افترض أنك بحاجة إلى استخدم ()LookupVacationBenefit من وحدة ثالثة لا تملك "كائن موظف" لكنها تملك تاريخ توظيف وتصنيف عمل. فجأة ()LookupVacationBenefit تبدو أقل ود وغير راغبة بالارتباط مع الوحدة الجديدة.

بالنسبة للوحدة الثالثة لتستخدم ()LookupVacationBenefit, عليها أن تعرف صف الموظف. يمكنها أن تصنع "دمية" (نموذج) من كائن الموظف لديه حقلان فقط، لكن هذا يتطلب معرفة داخلية ب ()LookupVacationBenefit, أعني أنها تستخدم هذين الحقلين. حل كهذا لا بد أن يكون حل أخرق وبشع.

سيكون الخيار الثاني أن تعدّل LookupVacationBenefit() بحيث تأخذ تاريخ التوظيف و تصنيف العمل بدلاً من موظف. في كلا الحالين، بانت الوحدة الاصلية أقل مرونة مما كانت تبدو عليه في البداية.

النهاية السعيدة للقصّة هي أن الوحدة غير الودودة يمكن أن تصنع صداقات إذا رغبت بأن تكون مرنة-في هذه الحالة، بتغييرها لتأخذ تاريخ التوظيف وتصنيف العمل بالتخصيص بدلاً من موظف.

باختصار، كلما كان أسهل للوحدات الأخرى أن تستدعي وحدة، كلما كان الاقتران أضعف. وهذا جيد لأنه أكثر مرونة وقابلية للصيانة. خلال إنشاء بنيان النظام، جزّء البرنامج على طول خطوط الاتصالية الداخلية الصغرى. إذا كان البرنامج قطعة خشب، ربما تود أن تجرب فصله باستخدام "قطاعة خشب".

أنواع الاقتران

هاهنا الأنواع الأكثر شيوعاً للاقتران التي ستصادفها.

اقتران "وسيط البيانات البسيط" تكون الوحدتان مقترنتين بـ "وسيط بيانات بسيط" إذا كانت كل البيانات الممرّرة فيما بينهما من أنماط البيانات الأساسية وكل البيانات تُمرر خلال قائمة وسطاء. هذا النوع من الاقتران عادي ومقبول.

اقتران "الكائن البسيط" تقترن الوحدة بـ"كائن بسيط" مع كائن إذا كانت تنسخ ذاك الكائن. هذا النوع من الاقتران ممتاز.

اقتران "الكائن الوسيط" تكون الوحدتان مقترنتين بـ"كائن وسيط " مع بعضهما إذا كان الكائن1 يتطلب أن يمرر له الكائن2 الكائن3.[1] هذا النوع من الاقتران أشد من حالة: الكائن 1 يتطلب أن يمرر له الكائن 2 أنواع البيانات الأساسية؛ لأن الحالة الأولى تتطلب أن يعرف الكائن 2 الكائن 3.

الاقتران المعنوي النوع الأكثر مكراً من الاقتران يحدث عندما لا تقوم وحدة باستخدام بعض العناصر القواعدية لوحدة أخرى بل تستخدم بعض المعرفة المعنوية عن أعمال الوحدة الداخلية. إليك بعض الأمثلة:

- الوحدة 1 تمرر راية تحكم إلى الوحدة 2 والتي تخبر الوحدة 2 ما عليها فعله. هذه الطريقة تتطلب أن تتدخل الوحدة 1 في الاشـغال الداخلية للوحدة 2. أعني ما الذي سـتفعله الوحدة 2 براية التحكم. إذا

[1] **أضف إلى معلوماتك** في اللغة العربية إذا لم تظهر حركة الفاعل والمفعول به، فالأول ترتيبا فاعل. مثلاً: ضرب مجدي حسني، مجدي الفاعل وحسني المفعول به

"عرفت" الوحدة 2 نوع بيانات محدّد لرايات التحكم (نمط معدود أو كائن)، ربما يكون هذا الاستخدام حسن.

- الوحدة 2 تستخدم بيانات شاملة بعد أن تكون هذه البيانات قد عدلت من قبل الوحدة 1. هذه الطريقة تتطلب أن تعتبر الوحدة 2 أن الوحدة 1 قد عدلت البيانات بالطريقة التي تطلبها الوحدة 2. وأن الوحدة 1 قد استدعيت في الوقت المناسب.
- واجهة الوحدة 1 تقرر أن إجرائيتها Module1.Initialize() ينبغي أن تُستدعى قبل أن تُستدعى Modul1.Routine(). الوحدة 2 تعرف أن Module1.Routine() تستدعي Module1.Initialize() بكل الأحوال، لذ تقوم بنسخ الوحدة 1 و تستدعي Module1.Routine() بدون أن تستدعي Module1.Initialize() قبلها.
- تمرر الوحدة 1 كائن إلى الوحدة 2. بسبب أن الوحدة 1 تعرف أن الوحدة 2 تستخدم فقط ثلاثة من طرق الكائن السابقة، تقوم بنسخ الكائن جزئياً-مع البيانات المحددة التي تحتاجها تلك الطرق الثلاثة.
- الوحدة 1 تمرر "كائن أب" إلى الوحدة 2. لأن الوحدة 2 تعرف أن الوحدة 1 مررت فعلياً "كائن ابن"، تقوم بتحويل قسري للكائن الأب إلى كائن ابن وتقوم باستدعاء الطرق الخاصة بالكائن الابن.

الاقتران المعنوي خطير لأن تغيير الشفرة في الوحدة المستخدَمة يمكن أن يعطل الشفرة في الوحدة المستخدِمة بطرق غير قابلة للاكتشاف من قبل "المترجم" نهائياً. عندما تتعطل الشفرة بطريقة كهذه، فإنّها تتعطل بطرق دقيقة تبدو غير مرتبطة بالتغييرات التي تمت في الوحدة المستَخدَمة، والتي تجعل من "التصحيح" أمراً عبثياً.

الغاية من الاقتران الضعيف هي أنّ الوحدة الفعالة تؤمن مستوى إضافي من التجريد-فور انتهائك من كتابتها، يمكن أن تستعملها على المضمون. إنها تخفف من تعقيد البرنامج الكلي وتسمح لك بالتركيز على شيء واحد كل مرة. إذا كان استخدام الوحدة يتطلب منك أن تركز على أكثر من شيء في المرة الواحدة-المعرفة بأشغال الوحدة الداخلية، التعديل على البيانات الشاملة، آلية العملية غير المؤكدة-تتم خسارة القوة التجريدية وتضعف قدرة الوحدة على المساعدة في التحكم بالتعقيد أو تزول.



إنّ الصفوف والاجرائيات هي الأدوات الفكرية الاولى والرئيسية في تخفيف التعقيد. وإذا لم يجعلوا دورك أبسط، فإنّهم لا يقومون بدورهم.

البحث عن نماذج التصميم الشائعة[1]

تقدم نماذج التصميم جواهر الحلول "الجاهزة للاستخدام" التي يمكن أن تستخدم لحل العديد من المشاكل الأكثر شيوعاً في البرمجيات. بعض مشاكل البرمجيات تتطلب حلول مشتقة من القواعد الأولى. لكن معظم

[1] cc2e.com/0585

المشاكل مشابهة لمشاكل الماضي، وهذه يمكن أن تُحل باستخدام حلول مشابهة، أو نماذج. النماذج الشائعة تتضمن:

المحوّل والجسر والمزخرف والواجهة وطريقة المصنع والمراقب والوليد الواحد والاستراتيجية وطريقة القالب. كتاب "نماذج التصميم" الذي كتب بواسطة: ايريك غاما وريتشارد هيلم ورالف جونسون وجون فليسيدس (1995) هو الوصف النهائي لنماذج التصميم.

تقدّم النماذج عدة ميزات لا يقدمها التصميم "المخصص بالكامل":

النماذج تقلل التعقيد بتأمين تجريدات جاهزة للاستخدام إذا قلت، "تستخدم هذه الشفرة "طريقة المصنع" لتنشئ نسخ من الصفوف الأبناء،" سيفهم المبرمجون الآخرون في المشروع أن شفرتك تحتوي على مجموعة غنية جداً من العلاقات المتبادلة وبروتوكولات البرمجة، كل منها يتم استحضارها عندما تذكر نموذج التصميم "طريقة المصنع."

"طريقة المصنع" هي النموذج الذي يسمح لك باستنساخ أي صف ابن من صف أب محدّد بدون الحاجة إلى متابعة الصفوف الأبناء المستقلة في أي مكان غير طريقة المصنع. لنقاش جيد عن نموذج طريقة المصنع، انظر "ابدال الباني بطريقة المصنع" في إعادة التصنيع (فولير 1999). " Replace Constructor with Factory Method" in Refactoring(Fowler 1999).

لا يتوجب عليك أن تهجئ كل سطر من الشفرة للمبرمجين الاخرين كي يفهموا نهج التصميم الموجود في الشفرة.

النماذج تخفف الأخطاء بتوظيف تفاصيل لحلول شائعة تحتوي مشاكل تصميم البرمجيات على دقائق لا تظهر كلياً إلا عندما تُحل المشكلة مرة أو اثنتين (أو ثلاث مرات أو أربعة أو...). لأن النماذج تمثل طرق معيارية لحل المشاكل الشائعة، فهي تجسّد الحكمة المتراكمة منذ سنين من المحاولات لحل هذه المشاكل، وكذلك تجسّد التصحيحات لمحاولات خاطئة قام بها أناس في حلهم لتلك المشاكل.

وهكذا، فإنّ استخدام نماذج التصميم هو مشابه بالمفهوم لاستخدام مكتبة شفرة بدلاً من كتابة واحدة بنفسك. بالتأكيد، قام كل شخص منا بكتابة خوارزمية الترتيب السريع الخاصّة به عدة مرات، لكن كم هي الحالات الشاذة التي فيها سيكون اصدارك الخاص صحيح تماما من المرة الأولى؟ بشكل مشابه، العديد من مشاكل التصميم مشابهة لحدٍ كافٍ لمشاكل الماضي ومن الأفضل لك كلياً أن تستخدم حلاً مسبقاً لتصميم من أن تبتكر حلاً جديداً.

تقدم التصاميم قيمة ارشادية باقتراح بدائل التصميم يستطيع بسهولة المصمم المتآلف مع النماذج الشائعة أن يتصفح قائمة من التصاميم ويسأل " أي هذه التصاميم يناسب مشكلتي التصميمية؟" المرور بسرعة على مجموعة من البدائل المشابهة هي أسهل بشكل غير قابل للقياس من انشاء حلك التصميمي الخاص بدءا من الصفر. والشفرة الناتجة من نموذج شائع ستكون أيضاً أسهل فهماً للقارئ مما ستكون عليه شفرتك الخاصّة.

النماذج تيسّر التواصل بنقل حوار التصميم إلى مستوى أعلى بالإضافة إلى فائدة إدارة التعقيد، يمكن لنماذج التصميم أن تسرع حوار التصميم بالسماح للمصممين بالتفكير والمناقشة في مستوى يكون فيه حجم الوحدات المكونة للبرنامج أكبر (larger level of granularity).[1] إذا قلت "لا أستطيع التقرير إن كان ينبغي استخدام "المنشئ" أو "طريقة المصنع" في هذه الحالة،" تكون قد قمت بصفقة رابحة في التواصل ببضع كلمات-طالما أنك ومستمعيك متآلفين مع هذه النماذج. تخيّل كم من الوقت ستأخذ أكثر لتغوص إلى تفاصيل الشفرة لنموذج "المنشئ" و "طريقة المصنع" ثم تقارن وتغاير بين النهجين.

إذا لم تكن متآلفاً مع نماذج التصميم، يلخّص الجدول 5-1 بعض النماذج الأكثر شيوعاً ليشد انتباهك.

الجدول 5-1 نماذج تصميم شائعة

النموذج	الوصف
المصنع المجرد (Abstract Factory)	يدعم إنشاء مجموعات من الكائنات المترابطة بتحديد نوع المجموعة وليس الأنواع لكل الكائنات المحدّدة.
المحوّل (Adapter)	يحوّل واجهة صف إلى واجهة أخرى.
الجسر (Bridge)	يبني واجهة وتحقيق بطريقة تمكن لأي منهما أن يتغير بدون أن يتغير الآخر.
التركيب (Composite)	يتألف من كائن يحوي كائنات إضافية من نفس نوعه بحيث تستطيع شفرة الزبون (الصف الذي يتعامل معه) أن تتفاعل مع الكائن ذو المستوى الأعلى دون أن تعير انتباها لكل الكائنات التفصيلية.
المزخرف (Decorator)	يلحق مسؤوليات إلى كائن بشكل ديناميكي. بدون انشاء صفوف فرعية محددة لكل ترتيب محتمل للمسؤوليات.
الواجهة (Facade)	يؤمن واجهة ملائمة لشفرة لا يمكن بطريقة أخرى أن تعرض واجهة ملائمة.
طريقة المصنع (Factory Method)	تستنسخ الصفوف الأبناء من صف أب محدّد بدون الحاجة إلى مراقبة الصفوف الأبناء المستقلة في كل مكان الاّ في طريقة المصنع.
المكرر (Iterator)	كائن خادم يؤمن الوصول إلى كل عنصر في مجموعة ما بشكل متعاقب.
المراقب (Observer)	يحافظ على مجموعة كائنات متزامنة مع بعضها البعض بجعل كائن آخر مسؤول عن تنبيه كائنات المجموعة عن التغيرات بأي عضو من المجموعة.
الوليد الواحد (Singleton)	يقدم نفاذ شامل إلى صف ذو نسخة واحدة وفقط واحدة.
الاستراتيجية (Strategy)	تعرّف مجموعة من الخوارزميات أو الأنماط السلوكية التي يمكن أن تتبادل بين بعضها بشكل ديناميكي.
طريقة القالب (Template Method)	تعرف بنيان لخوارزمية لكن تترك بعض التحقيق التفصيلي للصفوف الفرعية.

[1] granularity: اصطلاح يشير إلى مستوى تفصيل الشفرة في سياق ما، كلما كبر كلما كان السياق اكثر تجريدا، وكلما قل كلما كنت تغوص في تفاصيل الشفرة

إذا لم تكن قد رأيت نماذج التصميم من قبل، ربما ستكون استجابتك للأوصاف في الجدول السابق "بالتأكيد، أعرف مسبقاً معظم هذه الأفكار." هذه الاستجابة جزء كبير من سبب كون نماذج التصميم قيّمة. النماذج مألوفة لمعظم المبرمجين المحنكين، وإعطاء أسماء معروفة لها يدعم التواصل الفعّال والفعلي المتعلق بها.

أحد الفخاخ الكامنة في النماذج "اجبار مناسبة" الشفرة لاستخدام النموذج. في بعض الحالات، تبديل الشفرة قليلا لتناسب نموذج معروف جيداً سيحسّن إمكانية فهم الشفرة. لكن إذا توجّب على الشفرة أن تتبدل كثيراً، اجباراً لها لتبدو كنموذج قياسي، يمكن أن يزيد التعقيد أحيانا.

فخ كامن آخر في النماذج هو "التهاب الميزات". استخدام تصميم لرغبة بتجربته بدلاً من كون النموذج حل تصميمي مناسب.

على كل، نماذج التصميم أداة قوية لإدارة التعقيد. بإمكانك قراءة توصيفات أكثر تفصيلاً في أي من الكتب الجيدة المرتبة في نهاية هذا الفصل.

استدلالات أخرى

وصفت الأقسام السابقة الاستدلالات الرئيسية في تصميم البرمجيات. التالي هو استدلالات أخرى ربما ليست مفيدة تماما في معظم الأحيان لكنها تبقى جديرة بالذكر.

توجه إلى التماسك القوي

ينشأ التماسك من التصميم المُهيكَل ويناقَش عادة في نفس سياق "الاقتران." يشير التماسك إلى أي مدى، كل الاجرائيات في الصف أو كل الشفرة في إجرائية، تدعم غرض مركزي-كم هو الصف مُرَكّز. الصفوف التي تحوي آليات عمل مترابطة بقوة توصف بأنها تمتلك تماسك قوي، وهدف الاستدلال جعل التماسك أقوى ما يمكن. التماسك أداة مفيدة في إدارة التعقيد لأنه كلما كانت الشفرة في الصف تدعم غرض مركزي، كلما كان أسهل لدماغك أن يتذكر كل شيء تفعله الشفرة.

قد كان التفكير في التماسك على مستوى الإجرائية استدلالا مفيداً لعقود ومازال مفيداً اليوم. في مستوى الصف، استدلال التماسك قد صنف عموماً ضمن الاستدلال الأعم "تجريدات مُعرَّفة جيدا،" والتي نوقشت مسبقاً في هذا الفصل وستناقش في الفصل 6. التجريدات مفيدة في مستوى الإجرائية، أيضاً، وتَرْسخ الفائدة أكثر باشراك التماسك في ذلك المستوى من التفصيل.

ابنِ هرميات

الهرمية هي بنيان معلومات طبقي، يقبع فيه التمثيل الأعم أو الأكثر تجرداً للمفاهيم في قمّة الهرمية، مع تمثيلات تزداد تفاصيل وتخصيص في طبقات الهرمية الدنيا. في البرمجيات، توجد الهرميات في هرميات الصف، وكما موضح في المستوى 4 في الشكل 5-2، في هرميات استدعاء الإجرائية أيضاً.

لقد كانت الهرميات أداة هامة لإدارة مجموعات معقدة من المعلومات لـ 2000 سنة على الأقل. استخدم ارسطو الهرمية لينظم مملكة الحيوان. يستخدم البشر عادة مخططات عامة لينظموا المعلومات المعقدة (مثل هذا الكتاب). وجد باحثون أن البشر عموماً يجدون أنّ الهرميات طريقة طبيعية لتنظيم المعلومات المعقدة. عندما يرسمون كائن معقد مثل البيت، يرسمونه بشكل هرمي. بداية يرسمون الشكل الخارجي للبيت، ثم الشبابيك والأبواب، ثم تفاصيل أكثر. لا يرسمون البيت حجرة حجرة أو خشبة خشبة أو مسمار مسمار (سيمون 1995).

الهرميات أداة مفيدة لإنجاز الالزام التقني الرئيسي للبرمجيات لأنها تسمح لك بالتركيز على طبقة واحدة فقط من التفصيل تهمك حالياً. لا تذهب التفاصيل بعيداً بشكل كامل، إنّها ببساطة تُدفع إلى طبقة أخرى بحيث يمكنك التفكير بها (التفاصيل) عندما تريد ذلك بدلاً من التفكير بكل التفاصيل في كل الوقت.

رسّم اتفاقيات الصفوف[1]

في مستوى تفصيلي أكثر، التفكير بواجهة كل صف على أنّها اتفاقية مع بقية البرنامج يمكن أن تثمر عن رؤى جيدة. بشكل قياسي، الاتفاقية هي شيء يشبه "إذا وعدت أن تقدم البيانات ض وظ وغ ووعدت أنهم يملكون المميزات أ وب وج، فإني أعدك بأن أنجز العمليات 1 و2 و3 ضمن الشروط 8 و9 و10." الوعود التي يقدمها زبائن الصف للصف تسمى عادة "الشروط المسبقة،" والوعود التي يقدمها الكائن للزبائن تسمى "الشروط اللاحقة."

الاتفاقيات مفيدة في إدارة التعقيد لأنّه، على الأقل نظرياً، يستطيع الكائن بأمان أن يهمل أي سلوك غير متفق عليه. وعملياً، هذه القضية أعقد بكثير.

عيّن المسؤوليات

استدلال آخر هو أن تفكّر ملياً كيف ينبغي أن تنسب المسؤوليات إلى الكائنات. سؤال "عمّ ينبغي أن يكون كل كائن مسؤولا" مشابه لسؤال "ما المعلومات التي ينبغي أن يخفيها الكائن،" لكني أعتقد أنّه (الثاني) يمكن أن ينتج أجوبة أوضح، والتي تعطي الاستدلال قيمة فريدة.

صمم الاختبار

1 **إشارة مرجعية** لمزيد حول الاتفاقيات، راجع "استخدم التأكيد على الوثيقة وتحقق من الشروط المسبقة والشروط اللاحقة" في القسم 8. 2.

إن عملية التفكير التي يمكن أن تسفر عن رؤى تصميم مثيرة للاهتمام هي بالسؤال كيف سيبدو النظام إذا قمت بتصميمه لتسهيل عملية الاختبار. هل تحتاج إلى فصل واجهة المستخدم عن باقي الشفرة بحيث يمكنك استعمالها بشكل مستقل؟ هل تحتاج إلى تنظيم كل نظام فرعي بحيث تُقلّل التبعيات على الأنظمة الفرعية الأخرى؟ يميل التصميم للاختبار إلى ظهور واجهات للصف أكثر رسمية، وهو أمر مفيد بشكل عام.

تجنب الفشل

كتب هنري بيتروسكي، أستاذ الهندسة المدنية، كتابًا مثيرًا للاهتمام بعنوان "نماذج التصميم: حالة من تواريخ الخطأ والحكم في الهندسة (بتروسكي 1994)، الذي يؤرخ تاريخ الفشل في تصميم الجسر. يقول بتروسكي أن العديد من أخطاء بناء الجسور الكبيرة حدثت بسبب التركيز على النجاحات السابقة وعدم النظر بشكل مناسب على اوضاع الفشل المحتملة. ويختتم بتروسكي: إن فشل مثل فشل بناء جسر تاكوما الضيق كان يمكن تجنبه لو درس المصممون بعناية احتمالات فشل بناء الجسر، وليس فقط نسخ سمات التصاميم الناجحة الأخرى من تصاميم الجسور الأخرى.

تجعل الهفوات الأمنية رفيعة المستوى من مختلف النظم المعروفة في السنوات القليلة الماضية من الصعب عدم الاتفاق على أننا يجب أن نجد طرقاً لتطبيق أفكار فشل تصميم بتروسكي على البرامج.

اختر وقت الربط بوعي[1]

وقت الربط يشير إلى وقت إسناد قيمة محدّدة إلى متحول. الشفرة التي تربط بوقت مبكر تميل إلى البساطة، لكنها أيضاً تميل إلى مرونة أقل. في بعض الأحيان تستطيع أن تحصل على رؤية تصميمية جيدة من طرح أسئلة مثل: ماذا إذا أسندت هذه القيم في وقت أبكر؟ ماذا إذا أسندت هذه القيم لاحقا؟ ماذا إذا هيئت هذا الجدول هاهنا في الشفرة؟ ماذا إذا قرأت قيمة هذا المتحول من المستخدم في وقت التنفيذ؟

اصنع نقاط مركزية للتحكم

يقول ابي. جي. بلاوغير أن اهتمامه الأساسي هو "مبدأ المكان الصحيح الوحيد-ينبغي أن يكون هناك مكان صحيح وحيد للبحث عن أي قطعة صعبة من الشفرة، ومكان صحيح واحد لتقوم بتغيير الصيانة المرجّحة " (بلاوغير 1993). يمكن أن يكون التحكم مركزيا في الصفوف والاجرائيات وقوالب "ما قبل المعالج" وملفات المكتبات المضمنة `#include files`-حتى الثوابت المسماة هي مثال عن النقطة المركزية للتحكم.

الفائدة المخففة للتعقيد هي: كلما قل عدد الأماكن التي تبحث فيها عن بعض الأشياء، يكون التغير أسهل وآمن.

[1] **إشارة مرجعية** لمزيد حول وقت الربط، راجع القسم 10. 6، "وقت الربط"

خذ بعين الاعتبار استخدام "القوة العمياء"1

أحد أدوات الاستدلال القوية هي القوة العمياء. لا تستخف بها. حل القوة العمياء الذي ينجح أفضل بكثير من حل ذكي لا ينجح. من الممكن أن تأخذ وقتاً طويلاً لتحصل على حل ذكي ينجح. بتصور تاريخ خوارزميات البحث، على سبيل المثال، أشار دونالد كوث إلى أنّه بالرغم من أنّ أول وصف لخوارزمية البحث الثنائي نشر في 1946، استغرقت 16 سنة أخرى ليقوم شخص ما بنشر خوارزمية بحثت بشكل صحيح في قوائم من كل الحجوم (كوث 1998). البحث الثنائي أكثر براعة، لكن البحث التعاقبي (القوة العمياء) عادة كاف.

ارسم مخططاً

المخططات هي الأخرى أداة استدلال قوية. الصورة تغني عن 1000 كلمة-في سياق واحد. إنك تريد بالواقع أن تسقط معظم الكلمات الألف بسبب إحدى غايات استخدام الصورة، وهي أن الصورة يمكن أن تمثل المشكلة في مستوى أعلى من التجريد. في بعض الأوقات تريد أن تتعامل مع المشكلة بالتفصيل، لكن في الأوقات الأخرى تريد أن تكون قادراً على العمل بعمومية أكبر.

حافظ على تصميمك وحدوياً

هدف الوحدوية هو جعل كل إجرائية أو صف مثل "الصندوق الأسود": نعرف ما يدخل ونعرف ما يخرج، لكن لا نعرف ما يجري بداخله. صندوق أسود لديه مثل هذه الواجهة البسيطة وهكذا آلية عمل معرّفة بشكل جيد، يمكنك بدقة أن تتوقع الخرج الموافق من أجل أي دخل محدّد.

مفهوم الوحدوية مرتبط بإخفاء المعلومات والتغليف واستدلالات تصميم أخرى. لكن في بعض الأحيان التفكير بكيفية تجميع نظام من مجموعة من الصناديق السوداء يقدم رؤى لا يقدمها إخفاء المعلومات والتغليف، لذا المفهوم جدير بأن تضعه في جيب بنطالك الخلفي.

موجز لاستدلالات التصميم[2]

اليك موجز لاستدلالات التصميم الرئيسية:

- أوجد كائنات العالم الحقيقي
- شكّل تجريدات ملائمة

[1] عندما تقع في الشك استخدم القوة العمياء- **باتلر لامبسون**

القوة العمياء مصطلح يشير إلى كتابة الشفرة بطريقة بسيطة تستفيد من قوة الحاسوب، دون القيام باي تحليل ذكي

[2] وأكثر اثارة للتنبيه، نفس المبرمج قادر تماما على القيام بنفس المهمة لوحده بطريقتين او ثلاثة، أحيانا بلا وعي، لكن في الغالبية العظمى ليبحث عن المتعة في التنوع، او ليقدم شكل مغاير أنيق. —ايه. آر. براون و دبليو. ايه. سامبسون

- غلّف تفاصيل التحقيق
- ورّث عند الإمكان
- خبئ الاسرار (إخفاء المعلومات)
- حدّد المناطق المرجحة للتغيير
- حافظ على الاقتران ضعيفا
- ابحث عن نماذج التصميم الشائعة
- الاستدلالات التالية مفيدة أحياناً هي الأخرى:
- توجّه إلى التماسك القوي
- ابن هرميات
- رسّم اتفاقيات الصفوف
- عيّن المسؤوليات
- صمّم للاختبار
- تجنّب الإخفاق
- اختر زمن الربط بوعي
- اصنع نقاط مركزية للتحكم
- خذ بعين الاعتبار استخدام القوة العمياء
- ارسم مخططا
- حافظ على تصميمك وحدوياً

توجيهات لاستخدام الاستدلالات

يمكن أن نتعلم نُهج تصميم البرمجيات من نُهج التصميم في حقول أخرى. أحد الكتب الاصيلة في استدلالات حل المشاكل كان كتاب ادجي. بوليا كيف تحلها (1957) `G. Polya's How to Solve it`. يركز نهج بوليا المعمّم في حل المشاكل على حل المشاكل في الرياضيات. يلخص الشكل 5-10 نهجه، وهو ملائم لملخص مشابه في كتابه (للتأكيد عل أهمية ملخصه)[1].

1- **افهم المشكلة**. يتوجب عليك أن تفهم المشكلة.

[1] cc2e.com/0592

ما هو المجهول؟ ما هي المعطيات؟ ما هي الشروط؟ هل من الممكن تحقيق الشروط؟ هل الشروط كافية لتحديد المجهول؟ أو هل هي غير كافية؟ أو متكررة؟ او متناقضة؟

ارسم شكل. اختر نمط كتابة مناسب. افصل بين الأجزاء المختلفة من الشرط. هل يمكن أن تكتبهم (الشروط)؟

2- **ابتكر مخطط**. اوجد الاتصالات بين المعطيات والمجهول. قد تُجبر على التفكير بمسائل مساعدة إذا لم تجد الاتصال بينها. ينبغي أن تأتي أخيرا بمخطط للحل.

هل رأيت المشكلة من قبل؟ أو هل رأيت نفس المشكلة بهيئة مختلة قليلا؟ هل تعرف مشكلة متعلقة بها؟ هل تعلم مبرهنة يمكن أن تكون مفيدة؟

انظر إلى المجهول! حاول أن تفكّر بمشاكل مألوفة لديها نفس المجهول أو واحد مشابه له. إذا كان لديك مشكلة مرتبطة بمشكلتك وهذه المشكلة قد تم حلها. هل يمكنك استخدامه؟ هل يمكنك أن تستخدم نتائجه؟ هل يمكنك أن تستخدم طريقته؟ هل ينبغي عليك أن تأتي ببعض العناصر المساعدة كي تجعل ذلك ممكناً؟

هل يمكنك أن تعيد قول المشكلة؟ هل يمكنك أن تعيد قولها بصياغة أخرى؟ عُد إلى التعاريف.

إذا لم تستطع حل المشكلة المطروحة، جرب أن تحل مشاكل متعلقة بها أولاً. هل تستطيع تخيّل مشكلة متعلقة بها أسهل منالاً؟ مشكلة أعم؟ مشكلة أخص؟ مشكلة مناظرة؟ هل يمكنك أن تحل جزء من المشكلة؟ حافظ فقط على جزء واحد من الشروط، وأسقط الجزء الاخر؛ كم هو بعيد تحديد المجهول، كيف يمكن أن يتغير؟ هل يمكنك أن تستنتج شيء مفيد من المعطيات؟ هل يمكنك أن تفكر بمعطيات أخرى مناسبة لتحديد المجهول؟ هل بإمكانك تغيير المجهول أو المعطيات، أو كلاهما إذا اضطررت، بحيث يكون المجهول والمعطيات أقرب إلى (نفسيهما) بعضهم البعض؟

هل استخدمت كل المعطيات؟ هل استخدمت كل الشروط؟ هل اخذت بحسبانك كل المفاهيم الجوهرية المتضمنة في المشكلة؟

3- **انجاز المخطط**. انجز مخططك.

انجاز مخططك للحل، تفحص كل خطوة. هل تستطيع أن ترى بوضوح أن الخطوة صحيحة؟ هل تستطيع أن تبرهن أنّها صحيحة؟

4- **انظر إلى الوراء**. افحص الحل.

هل تستطيع اختبار النتائج؟ هل تستطيع تفحص النقاش؟ هل تستطيع استنتاج النتيجة بشكل مختلف؟ هل تستطيع أن ترى النتيجة بلمحة خاطفة؟

هل تستطيع استخدام النتيجة، أو الطريقة في مشاكل أخرى؟

الشكل 5-10 طور ادجي. بوليا نهج لحل المشاكل في الرياضيات مفيد أيضاً في حل المشاكل في تصميم البرمجيات (بوليا 1957).

أحد التوجيهات الأكثر فعالية هو ألاّ تلتصق بنهج واحد، إذا لم ينجح تمثيل التصميم في لغة النمذجة الموحدة UML، اكتبه باللغة الإنكليزية (بلغتك). اكتب برامج اختبار صغيرة. جرّب نهج مختلف تماماً. فكّر بحل القوة العمياء. استمر بالتخطيط والرسم بقلم الرصاص، ودماغك سوف يتبعك. إذا فشل كل هذا امش بعيداً عن المشكلة. أعنى حرفيا أن تذهب لتتمشى، أو أن تفكّر بشيء آخر قبل العودة إلى المشكلة. إذا قدمت أفضل ما تستطيع "وعدت بخفي حنين"، فإن اخراج المشكلة من بالك لفترة غالباً سيعطي نتائج بسرعة أكبر من المثابرة المتواصلة.

لا يتوجب عليك أن تحل كل مشكلة التصميم دفعة واحدة. إذا علقت، تذكر وجود خيار ينتظر قرار، لكن لاحظ أنك لا تملك حتى الآن معلومات كافية لتحل هذه القضية الخاصة. لماذا تقاتل في طريقك خلال 20 بالمئة الأخيرة من التصميم مع أنها (القضية الخاصّة) ستسقط إلى مكانها بسهولة خلال مرة قادمة؟ لماذا تصنع قرارات سيئة معتمدة على خبرة محدودة بالتصميم مع أنك تستطيع صنع قرارات جيدة معتمدة على خبرة أكبر لاحقا؟ بعض الناس لا يرتاحون إذا لم ينهوا تماما كل ما يجب عليهم فعله بعد دورة تصميم، لكن بعد أن تكون انشأت عدة تصاميم بدون تحليل قضايا قبل أوانه، سيبدو طبيعياً أن تترك قضايا غير مدروسة حتى تمتلك معلومات أكثر (زانايسر 1992، بيك 2000).

4 .5 تطبيقات التصميم

ركّز القسم السابق على استدلالات متعلقة بواصفات التصميم-كيف تريد أن يبدو التصميم المكتمل. يشرح هذا القسم استدلالات تصميمية تطبيقية، خطوات يمكن أن تتخذها والتي غالباً ما تعطي نتائج جيدة.

كرّر

ربما خضت تجربة تعلمت فيها كثيرا عن كتابة برنامج وحلمت أن تتمكن من كتابته مرة أخرى، متزوداً ببصائر اكتسبتها من كتابته في المرة الأولى. تُطبق نفس الظاهرة على التصميم، لكن دورات التصميم أقصر والآثار مع سير التطور أكبر، لذا بإمكانك أن تتحمل نتائج الانعطاف في حلقة التصميم لعدة مرات فقط.



التصميم هو عملية تكرارية. أنت لا تذهب عادة من النقطة أ إلى النقطة ب وحسب؛ بل تذهب من أ إلى ب وتعود إلى أ.

طالما أنك تدور خلال التصاميم المنتخبة وتجرب نُهج مختلفة. ستشاهد المنظرين العالي المستوى والمنخفض المستوى. ستساعدك الصورة الكبيرة التي تحصل عليها من العمل في قضايا المستوى العالي في وضع تفاصيل منخفضة المستوى في الحسبان. والتفاصيل التي تحصل عليها من العمل في قضايا المستوى المنخفض ستقدم

أساساً واقعياً متينا لقرارات المستوى العالي. الشد والسحب بين اعتبارات المستوى العالي والمستوى المنخفض ديناميكي بأمان؛ إنه ينشأ تركيب مضغوط أكثر استقراراً من واحد مبني كلياً من الأعلى فنزولا أو من الأسفل فصعودا.

لدى العديد من المبرمجين-العديد من الناس، في هذا المطلب-مشكلة في الطواف بين الاعتبارات العالية المستوى والمنخفضة المستوى. إنّ التبديل من منظر للنظام إلى منظر آخر مجهد فكرياً، لكنه جوهري لإنشاء التصاميم الفعالة. إذا اردت الاطلاع على تمارين مسلية لتحسين مرونتك الفكرية، اقرأ Conceptual Blockbusting (Adams 2001)، الموضح في قسم "مصادر اضافية" في نهاية هذا الفصل.

عندما تنتهي من أول محاولة تصميم وتبدو جيدة إلى حد كاف، لا تتوقف! المحاولة الثانية تقريباً دائما أفضل، وأنت تتعلم أشياء في كل محاولة تمكنك من تطوير التصميم الكلي[1]. بعد تجريب ألف مادة مختلفة للمبة الانارة السلكية بدون أي نجاح، كما هو شائع، سئل توماس اديسون إذا كان يشعر أن وقته ضاع لأنه لم يكتشف اي شيء. "تفاهة،" زُعم أن اديسون أجاب. "أنا اكتشفت ألف شيء لا يعمل." في العديد من الحالات، حل المشكلة بنهج واحد سيقدم رؤى تمكنك من حل المشكلة باستخدام نهج اخر أفضل.

فرّق تَسُد

كما أشار ادجر ديكسترا، لا يمتلك أحد دماغ كبير كفاية ليحتوي كل تفاصيل برنامج معقد، وهذا ينطبق أيضاً على التصميم. قسّم البرنامج إلى مناطق مختلفة حسب احتياجاتك، ومن ثم عالج كل واحدة من هذه المناطق على حدة. إذا وصلت إلى نهاية ميتة في احدى المناطق، اعد الكرّة!

التنقية المتزايدة أداة قوية لإدارة التعقيد. كما نصح بوليا في حل المشاكل الرياضية، افهم المشكلة، ابتكر مخطط، انجز المخطط، ثم انظر إلى الوراء لترى كيف صنعت (بوليا 1957).

نهجي التصميم، من الأعلى فنزولاً ومن الأسفل فصعوداً

ربما تكون الطريقتين "الأعلى فنزولا" و"الأسفل فصعودا" قديمتين، لكنهما تقدمان رؤية قيّمة لإنشاء تصاميم غرضية التوجه. الأعلى فنزولا تصميم يبدأ من المستوى الأعلى من التجريد. تقوم بتعريف صفوف القاعدة وعناصر التصميم العامة. ومع تطور التصميم، تزيد مستوى التفصيل، معرفاً الصفوف المشتقة، والصفوف المتعاونة، وعناصر التصميم التفصيلية الأخرى

الأسفل فصعوداً تصميم يبدأ من الخصوصيات ويمشي باتجاه العموميات. إنه يبدأ قياسيا بتعريف الكائنات المادية ثم يعمم مجموعات من الكائنات وصفوف القاعدة من هذه الخصوصيات.

[1] **إشارة مرجعية** إعادة التصنيع هي طريقة امنة لتجريب بدائل مختلفة في الشفرة. لمزيد حول هذا، راجع الفصل 24، "إعادة التصنيع"

يجادل بعض الناس بعنف بأن البدء بالعموميات والاتجاه إلى الخصوصيات هو الافضل، والأخر يجادل أنه لا يمكنك حقيقة تعريف مبادئ التصميم العام حتى تستنبط التفاصيل المميزة. إليك وجهتي النظر.

حجة الأعلى فنزولا

المبدأ الأساسي وراء نهج الأعلى فنزولا هو فكرة أن دماغ الانسان يستطيع أن يركّز على كمية محدّدة وحسب من التفاصيل في المرة الواحدة. إذا بدأت بالصفوف العامة وحللتهم إلى صفوف أكثر خصوصية خطوة بخطوة، لن يُجبر دماغك على التعامل مع الكثير من التفاصيل معا.

عملية "فرّق تسد" هي تكرارية لعدة أسباب محسوسة. أولاً، لأنك عادة لا تتوقف بعد المستوى الأول من التحليل. ستتابع لعدة مستويات. ثانياً، لأنك لا تستوطن في آثار المحاولة الأولى. أنت تحلل البرنامج باتجاه واحد. في عدة نقاط من التحليل، ستتخذ خيارات تتعلق بطرق تقسيم النظم الفرعية، ورسم شجرة الوراثة، وتشكيل تركيبات من الكائنات. تأخذ خيار وترى ما الذي سيحدث. ثم ستبدأ مرة أخرى وتحللها بطريقة مختلفة وترى إن كان ذلك يعمل بطريقة أفضل. بعد عدة محاولات، ستمتلك عدة أفكار جيدة عن: ماذا يفي بالغرض ولماذا.

إلى أي مدى قمت بتحليل البرنامج؟ واصل التحليل حتى تبدو كتابة الشفرة في الخطوة التالية أسهل من التحليل. اعمل حتى يبلغ مستوى السهولة والوضوح الظاهر على التصميم حداً كافياً. عند تلك النقطة، تكون انتهيت. إذا لم يكن واضحاً، اعمل قليلا أكثر. إذا كان الحل يبدو الآن غامضا ولو قليلا، سيسبب معاناة لأي شخص يعمل عليه لاحقا.

حجة الأسفل فصعودا

في بعض الاحيان نهج الأعلى فنزولا يكون تجريدياً جداً لدرجة انه من الصعب البدء به. إذا احتجت أن تتعامل مع شيء ملموس أكثر، جرّب نهج التصميم الأسفل فصعودا. اسأل نفسك، "ما هي الأشياء التي أعرف أن هذا النظام يحتاج ان يفعلها؟ " بلا شك، بإمكانك أن تجيب على هذا السؤال. ربما ستعرّف عدة مسؤوليات منخفضة المستوى يمكن أن تسندها إلى صفوف محددة. على سبيل المثال، ربما تعرف أن النظام يحتاج أن يهيئ تقريراً معيناً، ويحسب معطيات لهذا التقرير، ويضع عناوينه في المركز، ويعرض التقرير على الشاشة، ويطبع التقرير باستخدام الطابعة، وإلى ما هنالك. بعد أن تعرّف عدة مسؤوليات منخفضة المستوى، ستبدأ عادة بالشعور بارتياح كاف لتلق نظرة إلى الاعلى ثانية.

في حالات أخرى، الواصفات الرئيسية لمشكلة النظام. تُملى من الاسفل. ربما يتوجب عليك أن تربط مع أجهزة عتاد صلب تُملي متطلبات واجهتها مقدارا ضخما على تصميمك.

إليك بعض الأمور لتبقيها في دماغك عندما تقوم بالتركيب من الأسفل فصعوداً:

- اسال نفسك ما هي الأشياء التي أعرف أن النظام يحتاج أن يفعلها

- عرّف مسؤوليات وكائنات محددة من ذلك السؤال.
- عرف كائنات عامة، وجمّعهم باســتخدام تنظيمات الأنظمة الفرعية أو الحزم أو التركيبات داخل الكائنات أو الوراثة، اي واحدة أنسب.

تابع إلى المستوى القادم صعودا، او ارجع إلى القمّة وحاول ثانية أن تعمل باتجاه النزول.

في الحقيقة لا يوجد جدال

الاختلاف الرئيسي بين استراتيجيتي الأعلى فنزولاً والأسفل فصعوداً هو أن احداهما استراتيجية تحليلية والأخرى تركيبية. واحدة تبدأ من المشكلة العامة وتجزئها إلى قطع قابلة للإدارة، والأخرى تبدأ من قطع قابلة للإدارة ثم تبني حل عام. كل النهجين يمتلكان نقاط قوة وضعف علينا أن نأخذها بعين الاعتبار عندما نطبقهما على مشاكل التصميم.

قوة تصميم الأعلى فنزولاً أنّه سهل. النّاس جيدون بتكسير شيء ما كبير إلى مكونات أصغر، والمبرمجون جيدون بشكل خاص بذلك.

قوة أخرى لتصميم الأعلى فنزولا أنه بإمكانك تأجيل تفاصيل البناء. مذ أن الأنظمة غالباً ما تكون قلقة بشأن التغيرات في تفاصيل البناء (على سبيل المثال، التغيرات في هيكلة الملف أو هيئة التقرير)، فإنه من المفيد أن تعلم في مرحلة مبكرة أن هذه التفاصيل ينبغي أن تُخفى في صفوف في أسفل الهرمية.

إن واحدة من قوى نهج الأسفل فصعودا هي أنه عادة ينتج تعريف مبكر بآليات العمل المفيدة التي نحتاجها، والذي ينتج تصميم مصنّع جيدا ومدمج. إذا تم بناء أنظمة مشابهة مسبقاً، يمكّنك نهج الأسفل فصعودا من البدء بالتصميم الجديد بالنظر إلى قطع النظام القديم وسؤال نفسك "ما الذي بإمكاني أن أعيد استخدامه؟"

وأحد نقاط ضعف نهج التركيب الأسفل فصعودا أنه من الصعب أن يُستخدم حصريا (لوحده). معظم الناس يبدون أفضل عند أخذ مفهوم كبير وتكسيره إلى مفاهيم أصغر من اخذ مفاهيم صغيرة وصناعة مفهوم كبير واحد. إنها تشبه مسألة "جمعها بنفسك" القديمة: اعتقد أنني انتهيت، لذا لم لا تزال أجزاء بداخل الصندوق؟ لحسن الحظ، لا يتوجب عليك أن تستخدم نهج التركيب الأسفل فصعودا بشكل حصري.

يوجد ضعف آخر في استراتيجية التصميم الأسفل فصعوداً وهو أنه احياناً لا تجد نفسك متمكنا من بناء برنامج من قطع بدأت بها. لا تستطيع أن تبني طائرة من المكابح، وربما عليك أن تعمل في الاعلى قبل أن تعرف أنواع القطع التي ستحتاج في الاسفل.

لنلخص، الأعلى فنزولاً يميل إلى البدء بسيطاً، لكن في بعض الاحيان التعقيد منخفض المستوى يرتد إلى القمة، وهذه الارتدادات يمكن أن تجعل الاشياء أكثر تعقيدا مما تحتاجه فعليا لتكون. الأسفل فصعوداً يميل للبدء معقدا، لكن تعريف ذلك التعقيد في مرحلة مبكرة يؤدي إلى تصميم أفضل للصفوف عالية المستوى-إذا لم ينسف التعقيد النظام بأكمله من البداية!

في التحليل النهائي، التصميمين الأعلى فنزولاً والأسفل فصعوداً ليسا استراتيجيتان متنافستان-إنهما بتعاونهما مفيدان. التصميم عملية استكشافية، والذي يعني: لا حل مضمون أن يعمل في كل مرة. يحتوي التصميم عناصر "التجريب والحكم" والخطأ. جرّب تشكيلة من النُهج حتى تجد واحداً يعمل جيداً.

النماذج التجريبية[1]

في بعض الأحيان لا يمكنك أن تعرف حقا ما إذا كان التصميم سوف يعمل حتى تفهم بشكل أفضل بعض تفاصيل التنفيذ. قد لا تعرف ما إذا كان تنظيم قاعدة بيانات معين سيعمل حتى تعرف ما إذا كان سيفي بأهداف الأداء التي تريدها. قد لا تعرف ما إذا كان تصميم نظام فرعي معين سيعمل حتى تقوم بتحديد مكتبات واجهة المستخدم الرسومية المحددة التي ستعمل معها. هذه أمثلة على "العَوَص" الأساسي في تصميم البرمجيات – لا يمكنك تحديد مشكلة التصميم بشكل كامل حتى يتم حلها جزئيا على الأقل.

تقنية عامة لمعالجة هذه الأسئلة بتكلفة منخفضة هي نماذج تجريبية. كلمة "النماذج الأولية" تعني الكثير من الأشياء المختلفة لأشخاص مختلفين (مكونيل 1996) (McConnell 1996). في هذا السياق، النماذج الأولية تعني كتابة الحد الأدنى المطلق من الشفرة المهملة التي تحتاجها للإجابة على سؤال تصميمي معين.

النماذج الأولية تعمل بشكل سيء عندما لا ينضبط المطورون بكتابة *الحد الأدنى المطلق* من الشفرة اللازمة للإجابة على سؤال. لنفترض أن السؤال التصميمي هو "هل يمكن لإطار قاعدة البيانات الذي اخترناه دعم حجم المعاملات الذي نحتاجه؟" أنت لا تحتاج إلى كتابة أي شفرة إنتاجية للإجابة عن هذا السؤال. حتى أنك لا تحتاج إلى معرفة تفاصيل قاعدة البيانات. تحتاج فقط إلى معرفة ما يكفي لتقريب مشكلة المساحة – عدد الجداول، وعدد الإدخالات في الجداول، وهلم جرا. يمكنك بعد ذلك كتابة شفرة نمذجة بسيطة جدا والتي تستخدم جداول مع أسماء مثل *جدول1* و*جدول2* و*عمود1* و*عمود2*، وتعبئة الجداول ببيانات غير مهمة، ثم تجري اختبار الأداء الخاص بك.

كما أن النماذج الأولية تعمل بشكل سيء عندما يكون السؤال التصميمي غير محدد بما فيه الكفاية. سؤال تصميمي مثل "هل سيعمل إطار قاعدة البيانات هذا؟" لا يوفر ما يكفي للاتجاه إلى النماذج. سؤال تصميمي مثل "هل سيدعم إطار قاعدة البيانات هذا 1000 معاملة في الثانية في ظل الافتراضات (س) و (ع) و (ص)؟" يوفر أساسا أكثر صلابة للنماذج الأولية.

ينشأ الخطر النهائي للنماذج الأولية عندما لا يتعامل المطورون مع الشفرة على أنها شفرة مهملة. لقد وجدت أنه ليس من الممكن للناس كتابة الحد الأدنى المطلق من الشفرة للإجابة على سؤال إذا كانوا يعتقدون أن الشفرة

[1] cc2e.com/0599

سوف تُستخدم في نهاية المطاف في النظام الإنتاجي. وينتهي بهم المطاف إلى تنفيذ النظام بدلا من النماذج الأولية. من خلال تبني الموقف أنه بمجرد الإجابة على السؤال سيتم التخلص من الشفرة، يمكنك تقليل هذا الخطر. إحدى الطرق لتجنب هذه المشكلة هي إنشاء النماذج الأولية بتقنية مختلفة عن شفرة الإنتاج. يمكنك تصميم نموذج جافا في بايثون وانشاء نموذج لواجهة المستخدم في مايكروسوفت بوربوينت. إذا قمت بإنشاء نماذج أولية باستخدام تقنية الإنتاج، فإن المعيار العملي الذي يمكن أن يساعد هو فرض أن تكون أسماء الصفوف أو أسماء الحزم الخاصة بشفرة النموذج مسبوقة بكلمة "نموذج أولي". هذا على الأقل يجعل المبرمج يفكر مرتين قبل محاولة توسيع شفرة النموذج (ستيفنس 2003) (S 2003tephens).

إذا استخدمت بانضباط، النماذج هي أداة بمنزلة العمود الفقري لدى المصمم لمكافحة عَوَص التصميم. إذا استخدمت دون انضباط، تضيف النماذج بعض العوص الخاص بها.

التصميم التعاوني [1]

في التصميم، رأسان غالبا ما يكونان أفضل من واحد، سواء كان يتم تنظيم هذين الرأسين بشكل رسمي أو بشكل غير رسمي. يمكن أن يتخذ التعاون أي من عدة أشكال:

- يمكنك الدخول بشكل غير رسمي إلى مكتب زميل في العمل وتسأله لتبادل بعض الأفكار.
- تجلس أنت وزميلك معا في قاعة المؤتمرات وتقومان برسم بدائل التصميم على لوح المعلومات.
- تجلس أنت وزميلك في العمل معا أمام لوحة المفاتيح وتجريا تصميم تفصيلي في لغة البرمجة التي تستخدمها، أي أنه يمكنك استخدام برمجة مزدوجة، الموصوفة في الفصل 21 "البناء التعاوني".
- يمكنك جدولة اجتماع لمناقشة أفكارك التصميمية مع واحد أو أكثر من زملاء العمل.
- يمكنك جدولة عملية بحث رسمية مع كل الهيكل الموصوف في الفصل 21.
- لا تعمل مع أي شخص يمكنه مراجعة عملك، لذلك يمكنك القيام ببعض الأعمال الأولية، ووضعها في درج، والعودة إليها بعد أسبوع. تكون قد نسيت بما فيه الكفاية لتكون قادر على إعطاء نفسك مراجعة جيدة إلى حد ما.
- تسأل شخص ما خارج شركتك للحصول على المساعدة: أرسل الأسئلة إلى منتدى متخصص أو مجموعة أخبار.

إذا كان الهدف هو ضمان الجودة، فإنني أميل إلى التوصية بممارسة المراجعة الأكثر تنظيما، وعمليات البحث الرسمية، للأسباب الموصوفة في الفصل 21. ولكن إذا كان الهدف هو تعزيز الإبداع وزيادة عدد بدائل التصميم

[1] **اشارة-مرجعية** لمزيد من التفاصيل حول التطوير التعاوني، انظر الفصل 21 "البناء التعاوني".

المتولدة، وليس فقط لإيجاد الأخطاء، فإن النهج الأقل تنظيما يعمل بشكل أفضل. بعد أن تستقر على تصميم معين، قد يكون التحول إلى بحث أكثر رسمية مناسبا، تبعا لطبيعة مشروعك.

ما هي كمية التصميم الكافية؟[1]

في بعض الأحيان يتم فقط رسم أدنى المخططات المعمارية قبل البدء بالترميز. وفي أحيان أخرى، تقوم الفرق بإنشاء تصاميم على المستوى من التفصيل الذي يصبح فيه الترميز عملية آلية في معظمها. ما هو مقدار التصميم الذي يجب عليك القيام به قبل البدء في الترميز؟

وهناك مسألة ذات صلة وهي كم ستجعل التصميم رسمياً. هل تحتاج إلى مخططات تصميم رسمية ومصقولة، أم أن التقاطات كاميرا رقمية لبعض الرسومات على لوح الكتابة تكون كافية؟

تقرير كم مقدار التصميم الذي يجب القيام به قبل البدء في الترميز الكامل ومدى الرسمية التي يجب استخدامها في توثيق هذا التصميم هو علم متقن جدا. وينبغي الأخذ بعين الاعتبار خبرة الفريق، والعمر المتوقع للنظام، والمستوى المطلوب من الموثوقية، وحجم المشروع والفريق. ويلخص الجدول 5-2 كيفية تأثير كل من هذه العوامل على منهج التصميم

الجدول 5-2 رسمية التصميم ومستوى التفصيل المطلوب

العامل	مستوى التفصيل المطلوب في التصميم قبل البناء	رسمية التوثيق
فريق التصميم / البناء لديه خبرة عميقة في مجال التطبيقات.	منخفض التفاصيل	رسمية منخفضة
فريق التصميم / البناء لديه خبرة عميقة ولكن عديم الخبرة في مجال التطبيقات.	متوسط التفاصيل	رسمية متوسطة
فريق التصميم / البناء ليس خبيرا	متوسطة إلى عالية التفاصيل	رسمية منخفضة – متوسطة
فريق التصميم / البناء لديه دورة تغير معتدلة إلى مرتفعة.	متوسط التفاصيل	—
التطبيق حرج بالنسبة للأمان	عالي التفاصيل	رسمية عالية
التطبيق حرج بالنسبة للمهمة	متوسط التفاصيل	رسمية متوسطة – عالية
المشروع صغير.	منخفض التفاصيل	رسمية منخفضة
المشروع كبير.	متوسط التفاصيل	رسمية متوسطة
من المتوقع أن يكون للبرنامج عمر قصير (أسابيع أو أشهر).	منخفض التفاصيل	رسمية منخفضة
من المتوقع أن يكون للبرنامج عمر طويل (أسابيع أو أشهر).	متوسط التفاصيل	رسمية متوسطة

وقد يكون هناك عاملان أو أكثر من هذه العوامل تدخل في أي مشروع محدد، وفي بعض الحالات قد توفر العوامل نصائح متناقضة. على سبيل المثال، قد يكون لديك فريق ذو خبرة عالية يعمل على برنامج حرج بالنسبة

[1] نحن نحاول حل المشكلة عن طريق الإسراع في عملية التصميم بحيث يتم ترك وقت كافي في نهاية المشروع للكشف عن الأخطاء التي ارتكبناها لأننا أسرعنا في عملية التصميم. – جلينفورد مايرز (Glenford Myers)

للأمان. في هذه الحالة، ربما تخطئ إذا اخترت المستوى الأعلى من تفاصيل التصميم والرسمية. في مثل هذه الحالات، سوف تحتاج إلى وزن أهمية كل عامل والحكم على ما يهم أكثر.

إذا ترك مستوى التصميم لكل فرد، فعندما ينخفض التصميم إلى مستوى المهمة التي قمت بها من قبل أو إلى تعديل بسيط أو تمديد لهذه المهمة، ربما تكون على استعداد للتوقف عن التصميم والبدء بالترميز.

إذا كنت لا أستطيع أن أقرر مدى عمق التحقيق في التصميم قبل أن أبدأ الترميز، أميل إلى أن أخطئ واختار الدخول في مزيد من التفاصيل. تنشأ أكبر أخطاء التصميم من الحالات التي اعتقدت أنني ذهبت بها بعيدا بما فيه الكفاية، ولكن تبين فيما بعد أنني لم أذهب بعيدا بما فيه الكفاية لأتحقق إذا كان هناك تحديات تصميم إضافية. وبعبارة أخرى، فإن أكبر مشاكل التصميم لا تميل إلى أن تنشأ من المناطق التي عرفت أنها كانت صعبة وخلقت تصاميم سيئة لها، ولكن من المناطق التي اعتقدت أنها كانت سهلة ولم أخلق لها أي تصاميم على الإطلاق. أنا نادرا ما أواجه المشاريع التي تعاني من وجود الكثير من العمل التصميمي.

من ناحية أخرى، رأيت أحيانا المشاريع التي تعاني من الكثير من توثيق التصميم[1]. وينص قانون غريشام (Gresham's Law) على أن "النشاط المبرمَج يميل إلى إبعاد النشاط غير المبرمج" (سيمون 1965) (Simon 1965). إن الاندفاع المبكر لصقل وصف التصميم هو مثال جيد على ذلك القانون. أود أن أرى 80 في المئة من جهود التصميم تذهب إلى خلق واستكشاف العديد من بدائل التصميم و20 في المئة تذهب إلى خلق توثيق أقل صقلاً، من أن يكون 20 في المئة تذهب إلى خلق بدائل تصميم متوسطة و80 في المئة تذهب إلى صقل توثيق التصاميم، هذا ليس جيداً جداً.

متابعة عملك التصميمي[2]

النهج التقليدي لمتابعة عملك التصميمي هو ان تفصّل التصاميم في وثيقة تصميم رسمية. على أية حال، بإمكانك متابعة التصاميم بعدة طرق بديلة تعمل بنجاح في المشاريع الصغيرة والمشاريع غير الرسمية والمشاريع التي تحتاج طريقة "خفيفة نظيفة" لتسجل التصميم:

اكتب توثيق التصميم في الشفرة نفسها[3] وثق قرارات التصميم المفتاحية في تعليقات الشفرة، عادة توضع في ترويسة الصف او الملف. عندما تقرن هذا النهج بمستخلص توثيق مثل JavaDoc، فان هذا يضمن اتاحة

[1] لم أقابل أبداً إنساناً يريد قراءة 17,000 صفحة من التوثيق، وإذا كان هناك، لكنت قتلته لإخراجه من حوض الجينات البشرية. – **جوزيف كوستيلو** (Joseph Costello)

[2] cc2e.com/0506

[3] الاخبار السيئة هي انه، وحسب رأينا، لن نجد حجر الفيلسوف. لن نجد مطلقا عملية تسمح لنا بتصميم البرمجية بطريقة منطقية بشكل كامل. الاخبار الجيدة هي انه نستطيع ان نجعلها مزيفة. —ديفيد بارناس و بأول كليمينتس.

توثيق التصميم حالا للمبرمج الذي يعمل على قسم ما من الشفرة، وسيحسن فرصة حفاظ المبرمجين على توثيق التصميم محدث بشكل معقول لآخر تعديل.

تابع نقاشات التصميم وقراراته على ويكي احصل على نقاشات تصميمك بالكتابة، على ويكي المشروع (وهو، مجموعة صفحات ويب يمكن ان تُعدل بسهولة من قبل أي شخص في مشروعك باستخدام متصفح ويب). هذا سيتابع نقاشات تصميمك وقراراته بشكل آلي، وان كان مع زيادة كبيرة في الكتابة بدلا من الكلام. يمكنك أيضاً ان تستخدم ويكي لتلتقط الصور الرقمية لتدعم النقاشات الكتابية أو تضع روابط إلى مواقع ويب تدعم قرار التصميم أو "أوراق بيضاء" أو مواد أخرى[1]. هذه التقنية مفيدة بشكل خاص إذا كان فريق التطوير متوزعا جغرافيا.

اكتب ملخصات البريد الالكتروني بعد نقاش التصميم، عيّن شخصا ليكتب ملخصا للنقاش-خصوصا الذي تم تقريره-وارسله إلى فريق المشروع، أرشف نسخة من البريد الالكتروني في مجلد البريد الالكتروني العام للمشروع.

استخدم كاميرا رقمية أحد الحواجز الشائعة عن توثيق التصاميم هو الملل عند انشاء رسوميات التصميم بواسطة ادوات الرسم الشائعة. لكن خيارات التوثيق ليست محصورة بالحدين "التقاط التصميم بترميز رسمي مرتب بأناقة" أو "لا توثيق للتصميم بالمرة. "أخذ صور لرسومات اللوح الابيض بكاميرا رقمية ثم تضمين هذه الصور في المستندات التقليدية يمكن أن تكون طريقة منخفضة الجهد للحصول على 80 بالمئة من فائدة الحفاظ على رسومات التصميم بالقيام بحوالي 1 بالمئة من العمل المتطلب لو اخترت انشاء رسومات التصميم.

احفظ مخططات التصميم لا يوجد قانون ينص أن توثيق التصميم يجب ان يلائم القياس المعياري "letter" للورق. إذا كنت تقوم برسومات التصميم على ورق مخططات كبير، ببساطة يمكنك أرشفتها في مكان مناسب- أو، أفضّل أكثر، الصق المخططات على الجدران حول منطقة المشروع بحيث يسهل على الناس الرجوع إليها وتحديثها عند الحاجة.

استخدم بطاقات (صف – مسؤولية – معاون)[2] بديل آخر منخفض التقنية لتوثيق التصاميم أن تستخدم بطاقات الفهرسة. على كل ورقة، يكتب المصممون اسم الصف ومسؤولياته والأعوان (الصفوف الاخرى التي تعمل سويا مع الصف). ثم تعمل مجموعة التصميم على الاورق حتى يصبحوا راضين بأنهم صنعوا تصميماً جيداً.

[1] الورقة البيضاء: هي تقرير رسمي يتضمن معلومات عن قضية هامة

cc2e.com/0513 2

عند هذه النقطة، ببساطة احفظ البطاقات للرجوع اليها مستقبلا. بطاقات الفهرسة رخيصة وغير مخيفة وقابلة للتنقل، وتشجع على التفاعل في المجموعة (بيك 1991).

انشئ مخططات لغة النمذجة الموحدة وفق المستويات المناسبة للتفصيل احدى التقنيات الشائعة لتخطيط التصاميم تدعى لغة النمذجة الموحدة (UML)، والتي عُرّفت من قبل مجموعة ادارة الاغراض (فولير 2004). الشكل 5-6 المذكور سابقا في هذا الفصل يمثل تصميم مخطط صفوف بتلك اللغة. هذه اللغة تقدم مجموعة واسعة من التمثيلات الرسومية لوحدات التصميم والعلاقات. يمكنك ان تستخدم اصدار غير رسمي من هذه اللغة لتكتشف وتناقش مقاربات التصميم. ابدأ بالمخططات الصغرى واضف تفاصيل فقط بعد ان تكون قد افضيت من حل التصميم النهائي. لان هذه اللغة معيارية، فإنها تدعم الفهم الشائع لأفكار التصميم التواصلية ويمكنها ان تسرع عملية استعراض بدائل التصميم عندما يتم العمل في مجموعة.

هذه التقنيات يمكن ان تصنع تركيبات متعددة، لذا كن مرتاحا بمزج وانتقاء هذه النُهج وفق قاعدة "كل مشروع لوحده" او حتى في مناطق مختلفة من المشروع الواحد.

5.5 تعليقات على المنهجيات الشائعة

اشتهر تاريخ التصميم في البرمجيات بدفاعات متعصبة عن نُهج تصميم متصارعة بطريقة بريّة. عندما نشرت النسخة الأولى من *الشفرة الكاملة* بدايات التسعينات من القرن الماضي. المتعصبون للتصميم كانوا يدافعون عن الاهتمام بكافة تفاصيل التصميم قبل البدء بكتابة الشفرة. هذه النصيحة لا تعطي اي معنى.

وبينما انا اكتب هذه النسخة في منتصف العشر الأول من الالفية الثالثة، بعض المعلمين الروحيين البرمجيين كانوا يتجادلون بخصوص عدم القيام بأي تصميم على الاطلاق[1]. "Big Design Up Front is BDUF" (التصميم الكبير من البداية) هم قالوا "إن BDUF سيء. من الأفضل لك ان لا تقوم باي تصميم قبل البدء بكتابة الشفرة!"

في غضون عشر سنوات تأرجح بالبندول (من "تصميم كل شيء" إلى "تصميم لا شيء." لكن البديل ل BDUF ليس "لا تصميم من البداية"، لكن تصميم قليل من البداية Little Design Up Front (LDUF) او تصميم كاف من البداية Enough Design Up Front-ENUF.

[1] الناس الذين يقدمون الوعظ عن تصميم البرمجيات على انه اجراء عقابي يصرفون كمية معتبرة من الطاقة لجعلنا كلنا نشعر اننا مجرمون. لا نستطيع ابدا ان نكون مهيكلين كفاية او غرضيي التوجه كفاية لإنجاز النيرفانا خلال فترة الحياة هذه. كلنا ندور حول نوع من الخطيئة الأصيلة لأننا تعلمنا القواعد في عمر حساس. لكن رهاني ان معظمنا هم مصممون أفضل مما سيعترف به المتصوفون للأبد. --ابي. جي. بلاوغير

متى تقول أن هذه الكمية كافية؟ هذا محاكمة، ولا أحد يستطيع أن يقوم بهذا على نحو تام. لكن طالما أنك لا تستطيع أن تعرف المقدار الصحيح والدقيق من التصميم بثقة، فإن مقدارين من التصميم من المضمون أنهما خاطئين في كل المرات: تصميم كل التفاصيل إلى آخرها وعدم تصميم أي شيء مطلقا. كلا الموقفين الذين تم الدفاع عنهما من قبل المتطرفين لكلا النهايتين في المقياس، أصبحا الموقفين الوحيدين الذين يكونان دائما خطأ!

كما يقول بلاوغير، "كلما كنت جازما أكثر بتطبيق طريقة تصميم، كلما قل عدد مشاكل الحياة التي سوف تحلها" (بلاوغير 1993). عامل التصميم على أنه عملية استكشافية وغامضة وعويصة. لا تقعد بعد اول تصميم يخطر لك. تعاون وكافح من اجل البساطة. اصنع النماذج الأولية عندما تحتاجها. كرر وكرر وكرّر مجددا. سوف تكون سعيدا بتصاميمك.

مصادر إضافية[1]

تصميم البرمجيات حقل غني بمصادر غزيرة. التحدي هنا هو تحديد أي المصادر سيكون أكثر نفعا. هنا بعض الاقتراحات

تصميم البرمجيات، بالعموم

Weisfeld, Matt. The Object-Oriented Thought Process, 2d ed. SAMS, 2004. هذا كتاب سهل المنال، يقدم البرمجة غرضية التوجه. إذا كنت متآلفا من قبل مع البرمجية غرضية التوجه، ربما ستريد كتاب أكثر تقدما، لكن إذا كانت قدماك مبللتان فقط بغرضية التوجه، هذا الكتاب يقدم المفاهيم الغرضية التوجه الأساسية، متضمنا الكائنات والصفوف والواجهات والوراثة وتعدد الاشكال والتحميل الزائد والصفوف المجردة والتجميع-والارتباط-(aggregation & association) والبواني/الهوادم والاستثناءات واشياء أخرى.

Riel, Arthur J. Object-Oriented Design Heuristics. Reading, MA: Addison-Wesley, 1996. هذا الكتاب سهل القراءة ويركز على التصميم في مستوى الصف.

Plauger, P. J. Programming on Purpose: Essays on Software Design. Englewood Cliffs, NJ PTR Prentice Hall, 1993. لقد اقتنيت تلميحات عن تصميم البرمجيات الجيد من قراءة هذا الكتاب بمقدار ما اقتنيت من الكتب الأخرى التي قرأتها. بلاوغير متمكن جدا في مجموعة كبيرة من نُهج التصميم، انه واقعي، وانه كاتب عظيم.

[1] cc2e.com/0520

Meyer, Bertrand. Object-Oriented Software Construction, 2d ed. New York, NY: Prentice Hall PTR, 1997. قدم ماير دفاع قوي عن اعتناق البرمجة غرضية التوجه.

Raymond, Eric S. The Art of UNIX Programming. Boston, MA: Addison-Wesley, 2004. هذا منظر جيد ببحثه في تصميم البرمجيات عبر نظارات بلون يونكس. القسم 1. 6 هو ايجاز استثنائي، 12 صفحة تشرح 17 مبدأ تصميم مفتاحي في يونكس.

Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 2001.هذا الكتاب مقدمة محبوبة للتصميم غرضي التوجه في سياق "العملية الموحدة". وهو يناقش أيضاً التحليل غرضي التوجه.

نظرية تصميم البرمجيات

Parnas, David L., and Paul C. Clements. “A Rational Design Process: How and Why to Fake It.” IEEE Transactions on Software EngineeringSE-12, no. 2 (February 1986): 251–57. تشرح هذه المقالة التقليدية الهوة بين كيف تصمم البرامج فعليا وكيف تتمنى أحيانا ان يكونوا قد صمموا. الفكرة الرئيسية هي انه لا أحد ابدا يسير فعليا خلال عملية تصميم منطقية ومرتبة لكن بالتصويب على المنطقية والترتيب ستتجه إلى تصاميم أفضل في النهاية.

لا أعلم أي عرض شامل لإخفاء المعلومات. معظم كتب هندسة البرمجيات التخصصية تناقش هذه الفكرة بإيجاز، وكثيرا في سياق التقنيات غرضية التوجه. المقالات الاكاديمية الثلاث ل بارناس المرتبة في الأسفل هي الاب لأي عرض آخر لتلك الفكرة وهي ربما لا تزال أفضل مرجع في اخفاء المعلومات.

Parnas, David L. “On the Criteria to Be Used in Decomposing Systems into Modules." Communications of the ACM5, no. 12 (December 1972): 1053-58.

Parnas, David L. “Designing Software for Ease of Extension and Contraction.” IEEE Transactions on Software EngineeringSE-5, no. 2 (March 1979): 128-38.

Parnas, David L., Paul C. Clements, and D. M. Weiss. “The Modular Structure of Complex Systems.” IEEE Transactions on Software EngineeringSE-11, no. 3 (March 1985): 259-66

نماذج التصميم

Gamma, Erich, et al. Design Patterns. Reading, MA: Addison-Wesley, 1995. هذا الكتاب من عمل "عصابة الأربعة" هو أب لكتب نماذج التصميم.

Shalloway, Alan, and James R. Trott. Design Patterns Explained. Boston, MA: AddisonWesley, 2002. يحتوي هذا الكتاب على مقدمة لنماذج التصميم سهلة القراءة.

التصميم بالعموم

Adams, James L. Conceptual Blockbusting: A Guide to Better Ideas, 4th ed. Cambridge, MA: Perseus Publishing, 2001. على الرغم من أنه غير متخصص بتصميم البرمجيات، كُتب هذا الكتاب ليعلم طلاب الهندسة في جامعة ستانفورد التصميم. حتى وإن لم تصمم أي شيء من قبل، هذا الكتاب هو نقاش جذاب عن عمليات تفكير عبقرية. إنّه يتضمن العديد من التمارين في أنواع التفكير المطلوبة للتصميم الفعّال. ويحوي أيضاً قائمة مراجع مرتبة بشكل جيد في التصميم والتفكير العبقري. إذا كنت تحب حل المشاكل، ستحب هذا الكتاب.

Polya, G. How to Solve It: A New Aspect of Mathematical Method, 2d ed. Princeton, NJ: Princeton University Press, 1957. هذا نقاش حول الاستدلالات وحل المشاكل بالتركيز على الرياضيات لكنه قابل للتطبيق على تطوير البرمجيات. كان كتاب بوليا أول كتاب عن استخدام الاستدلالات في حل المشاكل الرياضية.

لقد حدّد بوضوح الفوارق بين الاستدلالات الفوضوية المستخدمة لاكتشاف الحلول، والتقنيات الأرتب المستخدمة لتجلّي الحلول حالما يتم اكتشافها (ربما تمر بحل المسألة لكن لا تعلم أنه الحل الذي تريده!). إنه ليس سهل القراءة، ولكن إن كنت مهتما بالاستدلالات، ستقرأه تدريجياً سواء أكنت تريد أو لا. أوضح كتاب بوليا أنّ حل المشاكل ليس نشاط حتمي والإخلاص إلى أية منهجية لوحدها هو مثل المشي وقدماك مكبلتان بالسلاسل. قامت مايكروسوفت في أحد المرات بإعطاء هذا الكتاب لكل مبرمجيها الجدد.

Michalewicz, Zbigniew, and David B. Fogel. How to Solve It: Modern Heuristics. Berlin: Springer-Verlag, 2000.هذا عرض مُحدّث لكتاب بوليا وهو أسهل قراءةً بكثير و يحتوي أيضاً بعض الأمثلة غير الرياضية.

Simon, Herbert. The Sciences of the Artificial, 3d ed. Cambridge, MA: MIT Press, 1996. يرسم هذا الكتاب الجذاب الفوارق بين العلوم التي تتعامل مع العالم الطبيعي (علم الاحياء، الجيولوجيا،...) والعلوم التي تتعامل مع العالم الاصطناعي الذي صنعه البشر (إدارة الأعمال، الهيكلة، علم الحاسوب). ثم يناقش مميزات علوم العالم الاصطناعي، وخصوصاً علم التصميم. لهجته أكاديمية وهو جدير بالقراءة لأي شخص عازم على مهنة في تطوير البرمجيات أو أي حقل "اصطناعي" آخر.

Glass, Robert L. Software Creativity. Englewood Cliffs, NJ: Prentice Hall PTR, 1995. هل تطوير البرمجيات خاضع للنظرية أكثر أم للتطبيق؟ هل هو إبداعي بشكل رئيسي أم حتمي؟ ماهي

درجة الجودة الفكرية التي يحتاجها تطوير البرمجيات؟ يحوي هذا الكتاب نقاشاً ممتعاً عن طبيعة تطوير البرمجيات وبتركيز خاص على التصميم.

Petroski, Henry. Design Paradigms: Case Histories of Error and Judgment in Engineering. Cambridge: Cambridge University Press, 1994. يخوض هذا الكتاب بشدة في حقل الهندسة المدنية (خاصة تصميم الجسور) ليبين حجته الرئيسية والتي هي: التصميم الناجح يعتمد على التعلم من الأخطاء السابقة على الأقل بمقدار اعتماده على النجاحات السابقة.

معايير

IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions. تحتوي هذه الوثيقة على معايير IEEE-ANSI حول توصيف التصميم البرمجي. إنها تصف ما ينبغي أن يُضمن في وثيقة التصميم البرمجي.

IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software Intensive Systems. Los Alamitos, CA: IEEE Computer Society Press. هذه الوثيقة هي دليل IEEE-ANSI لإنشاء مواصفات الهيكلية البرمجية

قائمة التحقق: التصميم في البناء[1]

عادات التصميم

- هل قمت بالتكرار، واخترت الأفضل من عدة محاولات بدلاً من المحاولة الأولى؟
- هل جربت تحليل النظام بعدة طرق مختلفة لترى أي الطرق ستعمل بشكل أفضل؟
- هل سرت في مشكلة التصميم بكلا الاتجاهين من الأعلى فنزولاً ومن الأسفل فصعوداً؟
- هل قمت بإنشاء نموذج أولي للأجزاء الخطيرة أو غير المألوفة من النظام، منشئاً المقدار الأصغري الثابت من الشفرة التي سترميه والضروري للإجابة على أسئلة تفصيلية؟
- هل تمت مراجعة تصميمك رسمياً أو غير رسمياً، من الآخرين؟
- هل قدت التصميم إلى نقطة يبدو فيها التحقيق (كتابة الشفرة) واضحا؟
- هل تابعت العمل التـصميمي با ستخدام تقنيات منا سبة مثل ويكي أو البريد الالكتروني أو عرض الـشرائح او الـصور الرقمية أو لغة النمذجة الموحدة أو بطاقات " صمم" أو التعليقات في الـشفرة نفسها؟

أهداف التصميم

[1] cc2e.com/0527

- هل عالج التصميم بكفاءة القضايا التي عُرّفت وأجلت في مرحلة الهيكلة؟
- هل قُسم التصميم إلى طبقات؟
- هل أنت راض عن الطريقة التي تم تحليل البرنامج بها إلى أنظمة فرعية وحزم وصفوف؟
- هل أنت راض عن الطريقة التي تم تحليل الصفوف بها إلى إجرائيات؟
- هل صُممت الصفوف على أساس التفاعل المتبادل الأصغري مع بعضها البعض؟
- هل صُممت الصفوف والأنظمة الفرعية بطريقة يمكن استخدامها في أنظمة أخرى؟
- هل سيكون البرنامج سهل الصيانة؟
- هل التصميم نحيل؟ هل كل أجزاء التصميم ضرورية بشدة؟
- هل استخدم التصميم التقنيات المعيارية وتجنب العناصر الغريبة والصعبة الفهم؟
- بعد كل شيء، هل ساعد التصميم في تقليل كلا التعقيدين الجوهري والعرضي؟

نقاط مفتاحية

- الالزام التقني الرئيسي في البرمجيات هو إدارة التعقيد. ويتم تسهيل هذا الالزام بشكل كبير بتصميم يركز على البساطة.
- تُنجز البساطة بطريقتين عامتين: تقليل كمية التعقيد الجوهري التي على دماغ أي شخص أن يتعامل معها في وقت واحد، والحفاظ على التعقيد العرضي من التكاثر غير الضروري.
- التصميم استكشافي. الإخلاص العقائدي إلى أية منهجية تؤذي الابداع وتؤذي برامجك.
- التصميم الجيد تكراري؛ كلما جربت احتمالات تصميم أكثر، كلما كان تصميمك النهائي أفضل.
- إخفاء المعلومات مفهوم قيّم بشكل خاص. السؤال "ماذا ينبغي أن أخفى؟" يحل العديد من قضايا التصميم الصعبة.
- الكثير من المعلومات المفيدة والممتعة حول التصميم متوفرة خارج هذا الكتاب وجهات النظر المقدمة هنا هي فقط قمة جبل الجليد.

الفصل 6

الصفوف الناجحة

المحتويات[1]

مواضيع ذات صلة

- التصميم في البناء: الفصل 5
- هيكلة البرمجيات: المقطع 3.5
- الإجرائيات عالية الجودة: الفصل 7
- عملية البرمجة باستخدام شفرة زائفة: الفصل 9
- إعادة التصنيع: الفصل 24

منذ فجر الحوسبة، فكّر المبرمجون بالبرمجة، على مستوى العبارات البرمجية. وخلال السبعينات والثمانيات بدأ المبرمجون بالتفكير بالبرمجة على مستوى الإجرائيات (`routines`). وفي القرن الحادي والعشرين، فكر المبرمجون بالبرمجة على مستوى الصفوف.



الصف عبارة عن تجمع بيانات وإجرائيات تتشارك مسؤولية محدّدة موحدّة. وأيضاً الصف من الممكن أن يعبر عن تجمع إجرائيات توفر مجموعة موحدّة من الخدمات، حتى لو لم تحوي على بيانات

[1] cc2e.com/0665

مشتركة. إنّ المفتاح الأساسي لتصبح مبرمج مُحترف، هو بزيادة حجم الجزء من البرنامج الذي يمكن تجاهله بشكل آمن أثناء العمل على قسم معين من الشفرة. حيث تعد الصفوف الأداة الأساسية لتنفيذ هذا الهدف.

يحتوي هذا الفصل على نصائح مُركّزة لإنشاء صفوف عالية الجودة. أما إذا كنت لا تزال مبتدئ في مفاهيم البرمجة غرضيّة التوجه (object-oriented)، فإنّ هذا الفصل يعدّ مستوى متقدم. كُن متأكداً من قراءتك للفصل الخامس " التصميم في البناء". ومن ثم ابدأ بالقسم 1.6 "أنواع البيانات المجردة"، لتسهل طريقك للأقسام الأخرى. أما إذا كنت على دراية بأساسيات الصف، فيمكنك تجاوز القسم 1.6 والبدء بقراءة القسم 2.6. يحوي قسم "المصادر الإضافية" في نهاية هذا الفصل على أسماء مراجع للقراءة التمهيدية والقراءة المتقدّمة والقراءة حول البرمجة بلغة محددة.

1.6 أساسيات الصف: أنواع البيانات المجردة

يُعرّف نوع البيانات المجردة (ADT - Abstract Data Types) بأنه مجموعة من البيانات والعمليات التي تعمل على ذاك النوع، تقوم العمليات بشيئين: وصف البيانات لبقية البرنامج والسماح لبقية البرنامج بتغير البيانات. تُستخدم كلمة "البيانات" في "نوع المعطيات المجردة" بشكل فضفاض. حيث أنه من الممكن أن يكون نوع البيانات المجردة عبارة عن نافذة رسومات مع كل العمليات التي تؤثر عليها، أو ملف والعمليات على هذا الملف، أو جدول معدلات التأمين والعمليات على هذا الجدول، أو أي شيء آخر.

إن فهم أنواع البيانات المجردة ضروري لفهم البرمجة غرضية التوجه[1]. وبدون فهم أنواع البيانات المجردة، فإن الصفوف التي يشكلها المبرمج هي عبارة عن "صفوف" فقط بالاسم. ولكن مع فهم أنواع البيانات المجردة، يمكن للمبرمجين إنشاء صفوف تكون أسهل للتنفيذ في البداية وأسهل للتعديل مع مرور الوقت.

عادةً، تزداد الرياضيات في كتب البرمجة عندما تصل إلى موضوع أنواع البيانات المجردة. حيث يميلون لاستخدام عبارات كالتالية "من الممكن فهم نوع البيانات المجردة كنموذج رياضي مع مجموعة من العمليات المحددة لهذا النموذج". توحي مثل هذه الكتب بأنك لم تستخدم أبدًا بشكل فعلي أي نوع من أنواع البيانات المجردة.

تفقد مثل هذه التفسيرات فعلًا الفهم الكامل لأنواع البيانات المجردة. إن أنواع البيانات المجردة مثيرة للاهتمام لأنه من الممكن استخدامها للتلاعب مع كيانات العالم الحقيقي بدلًا من التلاعب مع كيانات منخفضة المستوى. مثلاً، بدلًا من إدخال عقدة إلى قائمة مرتبطة، يمكنك إضافة خلية إلى جدول بيانات، أو إدخال نوع نافذه جديد إلى قائمة أنواع النوافذ، أو أضافة عربة ركاب أخرى إلى محاكاة قطار. استفد من نقطة القوة كونك قادر على العمل في مجال المشكلة بدلًا من العمل في مجال التنفيذ منخفض المستوى.

[1] **إشارة مرجعية**: يُعتبر التفكير حول أنواع البيانات المجردة أولًا، والتفكير حول الصفوف ثانياً، مثال عن البرمجة في لغة برمجة ما مقابل البرمجة في لغة برمجة معينة. انظر القسم 4-3 "موقعك في موجة التكنولوجيا"، والقسم 34-4 "برمج داخل لغتك وليس إليها".

أمثلة عن الحاجة إلى نوع البيانات المجردة

لنبدأ بمثال عن حالة معينه، والتي من الممكن أن نستخدم فيها نوع البيانات المجردة. وبعد التحدث عن هذا المثال سوف نتكلم عن تفاصيل أخرى.

لنفترض أنك تكتب برنامج عن التحكم بالنص على وحدة الخرج الشاشة، باستخدام مجموعة متنوعة من المحارف، والحجوم النقطية وسمات الخط (على سبيل المثال: ثخين أو مائل). حيث يتلاعب جزء من البرنامج بحجم خط النص. إذا كنت تستخدم نوع البيانات المجردة سيكون لديك مجموعة من الإجرائيات لتحرير الخط مع البيانات التي تعمل على هذه الإجرائيات- أسماء المحارف، والحجوم النقطية، وسمات الخط. مجموعة إجرائيات تحرير الخط والبيانات هي عباره عن نوع بيانات مجردة.

أما إذا لم تكن تستخدم نوع البيانات المجردة، فسوف تستخدم منهج مخصص لتغيير السمات الخطية للنص. على سبيل المثال إذا كنت ترغب بتغيير حجم خط النص إلى 12نقطة، والذي يكافئ ارتفاع 16بكسل، فإنه عليك استخدام الشفرة التالية:

```
currentFont.size = 16
```

أما إذا كنت قد أنشئت مكتبة من مجموعة من الإجرائيات، فمن الممكن أن تبدو الشفرة بالشكل التالي الأكثر قابلية للقراءة:

```
currentFont.size = PointsToPixels( 12 )
```

أو يمكنك استخدام اسم للسمة أكثر تحديداً، شيء من هذا القبيل:

```
currentFont.sizeInPixels = PointsToPixels( 12 )
```

ولكن من غير الممكن استخدام كلا "currentFont.sizeInPixels" و "currentFont.sizeInPoints" مع بعض، وذلك لأنه عند استخدام كلا النوعين من عناصر البيانات، سيؤدي هذا إلى عدم قدرة "currentFont" على معرفة أي نوع من عناصر البيانات سيُستخدم.

وإذا غيّرت حجم الخط في عدة أماكن من البرنامج، فإنك ستحصل على خطوط متماثله منتشرة على طول البرنامج. وإذا كنت بحاجة إلى إعداد سمة الخط العريض، فيمكنك استخدم الشفرة التالية، التي تحوي على العملية المنطقية "or" وعدد ثابت بالترميز الست عشري 0x02:

```
currentFont.attribute = currentFont.attribute or 0x02
```

وإذا كنت محظوظاً، سيكون لديك شيء أكثر وضوحاً من الشفرة السابقة، ولكن أفضل طريقة لكتابة الشفرة المطلوبة عن طريق المنهج التخصصي هو شيء من هذا القبيل:

```
currentFont.attribute = currentFont.attribute or BOLD
```

أو ربما شيء من هذا القبيل:

```
currentFont.bold = True
```

تكون المحدوديّة باستخدام حجم الخط، بأن شفرة المستخدم عليها أن تتحكم بعناصر البيانات بشكل مباشر. وهذا يحد من طريقة استخدام currentFont.

إذا كنت تبرمج بهذه الطريقة، فمن المحتمل أن يكون لديك أسطر مشابهة في عدّة أماكن من برنامجك.

فوائد أنواع البيانات المجردة

ليست المشكلة بأن المنهج التخصصي في البرمجة، هو عبارة عن أسلوب عملي سيء للبرمجة. المشكلة هي بأنه من الممكن تغيير هذه الطريقة بطريقة برمجية أفضل تُنتج الفوائد التالية:

يمكنك إخفاء تفاصيل التنفيذ يعني إخفاء معلومات نوع البيانات حول الخط، حيث أنه عندما يتغير نوع البيانات، فأنه يمكنك تغييره في مكان واحد فقط بدون التأثير على البرنامج بأكمله. على سبيل المثال، تغيير نوع البيانات (ثخانة الخط) من الدرجة الأول إلى الدرجة الثانية، سوف يُغير برنامجك في كل مكان تم تعيين الخط فيه ثخين، وليس في جزء واحد فقط. يحمي إخفاء المعلومات أيضاً بقية البرنامج إذا قررت أن تُخزن البيانات تخزيناً خارجياً، بدلاً من استخدام الذاكرة، أو أذا رغبت بإعادة كتابة جميع الإجرائيات لتحرير الخط في لغة برمجية أخرى.

لا تؤثر التغييرات على البرنامج بأكمله إذا كانت هناك حاجة بأن تصبح أنواع الخط أغنّى، وبإمكانها أن تدعم عدد أكبر من العمليات (مثل التبديل إلى القبعات الصغيرة، أو إلى الحروف الفوقية، أو إلى تشطيب الخط، وإلى آخره)، يمكنك تغيير البرنامج في مكان واحد. حيث لن يؤثر التغيير على بقية البرنامج.

يمكنك تشكيل واجهة غنية أكثر بالمعلومات مثلاً تبدو الشفرة (currentFont.size = 16) غامضة، لأنه من الممكن أن تكون 16 بكسل أو 16 نقطة. حيث لا يخبرك السياق لأي شيء ينتمي هذا الشيء. يسمح لك تجميع العمليات المتشابهة في نوع بيانات مجردة واحده، بتعريف واجهة كاملة مختصة بالنقاط، أو مختصة بالبكسلات، أو يسمح بالتفريق الواضح بين النوعين، بما يساعد على عدم الخلط بينهم.

تحسين الأداء أسهل إذا كنت ترغب بتحسين أداء أنواع الخط، فيمكنك إعادة كتابة الشفرة فقط عن طريق كتابة بضع إجرائيات، بدلاً من الخوض في كامل تفاصيل البرنامج.

البرنامج صحيح بشكل أكثر وضوحاً حيث يمكنك استبدال المهمة المملة للتحقق من صحة بعض العبارات مثل "currentFont.attribute = currentFont.attribute or 0x02 " بعملية أكثر سهولة للتحقق من صحة عبارة، مثل عملية الاستدعاء هذه

."currentFont.SetBoldOn() "

في الحالة الأولى، يمكنك أن تحصل على خطأ في اسم البنية، أو خطأ في اسم الحقل، أو خطأ في العملية (مثلا العملية المنطقية and بدلاً من or)، أو خطأ في قيمة الصفة (مثلا *0x20* بدلًا من *0x02*).

في الحالة الثانية، الخطأ الوحيد الذي من الممكن التفكير فيه في عملية الاستدعاء currentFont.SetBoldOn()، هو هل تم استخدام اسم خاطئ للإجرائية، و هكذا فمن السهل في هذه الحالة التحقق من صحة البرنامج.

يصبح البرنامج ذاتي التوثيق بشكل أكبر تستطيع تحسين بعض العبارات كالعبارة " currentFont.attribute or *0x02*"، عن طريق استبدال "*0x02*" ب "BOLD" أو أي شيء تُمثله "*0x02*"، ولكن هذا لا يمكن مقارنته مع إمكانية القراءة الكبيرة لاستدعاء إجرائية مثل "*currentFont.SetBoldOn()*" .

بيانات مثبتة

أجرى كل من وودفيلد ودونزمور وشين (Woodfield, Dunsmore, and Shen) دراسة، أجاب فيها طلاب الدراسات العليا في قسم علوم الحاسوب على أسئلة حول برنامجين: البرنامج الأول مُقسم على 8 إجرائيات، تُغطي الوظائف الرئيسية، أما البرنامج الثاني مُقسم على 8 إجرائيات لنوع البيانات المجردة (1981). حصّل الطلاب، الذين استخدموا برنامج نوع البيانات المجردة نقاطاً أكثر من الطلاب الذين استخدموا النسخة المتخصصة من البرنامج.

ليس من الضرورة أن تمرر البيانات عبر كل البرنامج في الأمثلة السابقة التي تم تقديمها، عليك أن تغير "currentFont" بشكل مباشر، أو تقوم بتمريره لكل إجرائية تعمل مع الخطوط. ولكن إذا استخدمت نوع البيانات المجردة، فإنه لن تكون هناك ضرورة لا لتمرير "currentFont" عبر البرنامج بالكامل، ولا لتضمين "currentFont" في البيانات العامة. حيث يحوي نوع البيانات المجردة على بنية تحوي على البيانات "currentFont". يتم الوصول إلى البيانات بشكل مباشر فقط عن طريق الإجرائيات، التي تحوي على جزء من نوع البيانات المجردة.

تستطيع العمل مع كيانات العالم الحقيقي، بدلاً من العمل مع هياكل تنفيذ منخفضة المستوى. حيث أنك تستطيع تعريف العمليات، التي تتعامل مع أنواع الخطوط، وبذلك يعمل الجزء الأكبر من البرنامج فقط مع الخطوط، بدلاً من العمل مع المصفوفات وتعريفات البنى، والقيم المنطقية True وFalse.

في هذه الحالة، لتعريف نوع البيانات المجردة، عليك أن تعرّف مجموعة من الإجرائيات للتحكم بالخطوط، ومن الممكن أن يكون لها الشكل التالي:

```
currentFont.SetSizeInPoints(sizeInPoints)
currentFont.SetSizeInPixels(sizeInPixels)
```

```
currentFont.SetBoldOn( )
currentFont.SetBoldOff( )
currentFont.SetItalicOn( )
currentFont.SetItalicOff( )
currentFont.SetTypeFace ( faceName)
```



من الممكن أن تكون الشفرة المكتوبة داخل هذه الإجرائيات قصيره، ومن الممكن أن تكون مشابهة للشفرة التي رأيتها سابقاً في هذا الفصل لحل مشكله أنواع الخطوط باستخدام المنهج التخصصي. ولكن يتجلى الاختلاف في أنك قمت بعزل العمليات المتعلقة بأنواع الخط في مجموعة من الإجرائيات. يزود هذا برنامجك، الذي يعمل مع أنواع الخطوط، بمستوى أفضل من التجريدية، ويعطيك طبقة من الحماية ضد التغيرات في عمليات الخط.

المزيد من الأمثلة حول أنواع البيانات المجردة

افترض أنك تقوم بكتابة برنامج يتحكم بنظام التبريد لمفاعل نووي. يمكنك أن تتعامل مع نظام التبريد كنوع بيانات مجردة، عن طريق تعريف العمليات التالية له:

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate( rate)
coolingSystem.OpenValve( valveNumber)
coolingSystem.CloseValve( valveNumber)
```

ستقوم البيئة المختصة بتحديد الشفرة المكتوبة، التي تقوم بتنفيذ كل من هذه العمليات. وبإمكان باقي البرنامج التعامل مع نظام التبريد عن طريق هذه التوابع، وليس عليه أن يقلق حول التفاصيل الداخلية لتنفيذيات بنى البيانات ومحدودات بنى البيانات والتغيرات وما شابه ذلك.

وهنا نستعرض المزيد من الأمثلة حول أنواع البيانات المجردة والعمليات المحتملة عليها:

خزان الوقود	الخلّاط	مثبت السرعة
املأ الخزان	تشغيل	تعيين السرعة
صرّف الخزان	إطفاء	الحصول على الإعدادات الحالية
احصل على سعة الخزان	تعيين سرعة	استئناف السرعة السابقة
احصل على حالة الخزان	شغل عملية الخلط	عطّل
	أطفئ عملية الخلط	
مكدس	**الضوء**	**القائمة List**
جهز المكدّس	تشغيل	جهز القائمة
ادفع عنصر في المكدّس	إطفاء	أدخل عنصر في القائمة
اسحب عنصر من المكدّس		احذف عنصر من القائمة

قراءة قمة المكدس		اقرأ العنصر التالي في القائمة
الملف	**اللائحة Menu**	**مجموعة من شاشات المساعدة**
افتح الملف	ابدأ بلائحة جديدة	إضافة موضوع المساعدة
اقرأ الملف	حذف قائمة	حذف موضوع المساعدة
اكتب في الملف	إضافة عنصر قائمة	تعيين موضوع المساعدة الحالي
تعيين موقع الملف الحالي	حذف عنصر لائحة	عرض شاشة المساعدة
اغلق الملف	فعّل عنصر لائحة	إزالة عرض شاشة المساعدة
	الغاء تفعيل عنصر قائمة	عرض فهرس المساعدة
المصعد	عرض القائمة	العودة إلى الشاشة السابقة
الصعود طابق واحد	إخفاء القائمة	
الهبوط طابق واحد	الحصول على خيار القائمة	**المؤشر**
الوصول إلى طابق محدد		نقل المؤشر إلى الذاكرة الجديدة
الإعلام عن الطابق الحالي		جهّز الذاكرة من المؤشر الحالي
العودة إلى الطابق الرئيسي		غير من حجم الذاكرة المحجوز

يمكنك أن تستخلص عدة توجيهات من دراسة هذه الأمثلة؛ تم وصف هذه التوجيهات في الأقسام الفرعية التالية:

ابنِ أو استخدم أنواع البيانات منخفضة المستوى النموذجية كأنواع بيانات مجردة، وليس كأنواع بيانات منخفضة المستوى تُركز معظم مناقشات أنواع البيانات المجردة على تمثيل أنواع البيانات منخفضة المستوى بأنواع البيانات المجردة. كما تستطيع أن ترى من الأمثلة، فإنه بإمكانك أن تمثل مكدس، قائمة، رتل، وعمليًا أي نوع بيانات نموذجي، كنوع بيانات مجردة. السؤال الذي عليك أن تسأله: "ماذا يمثل هذا المكدس، أو هذه القائمة، أو هذا الطابور؟" فإذا كان المكدس يمثل مجموعة من العمال، فيجب التعامل مع نوع البيانات المجردة كعمال وليس كمكدس. وإذا كانت القائمة تُمثل مجموعة من سجلات الفاتورة، فيجب التعامل مع سجلات الفاتورة، وليس مع قائمة. وإذا كان الرتل يمثل خلايا في جدول، فيجب أن تتعامل مع مجموعة من الخلايا بدلًا من التعامل مع عنصر عام في رتل. أي يجب عليك أن تتعامل مع أعلى مستوى ممكن من التجريدية.

تعامل مع العناصر الشائعة مثل الملفات كأنواع بيانات مجردة تحوي معظم اللغات على مجموعة من أنواع البيانات المجردة، التي من الممكن أن تكون متألف معها ولكن لم تفكر فيها كأنواع بيانات مجردة. حيث تُعتبر العمليات على الملفات بمثال جيد لهذا. اثناء الكتابة على القرص، فإن نظام التشغيل يقوم عنك بوضع رأس الكتابة /القراءة في عنوان فيزيائي محدد، ويقوم عنك بتخصيص قطاع جديد من القرص عند استنفاذك للقطاع القديم، ويقوم عنك بتفسير رموز الخطأ المخفية. يزود نظام التشغيل مستوى أول من التجريدية، وأنواع البيانات المجردة لهذا المستوى. أما لغات البرمجة عالية المستوى تزود مستوى ثاني من التجريدية، وأنواع البيانات المجردة لهذا المستوى الأعلى. تقوم اللغة عالية المستوى بحمايتك من فوضى التفاصيل، كتوليد نظام

التشغيل استدعاءات لعمليات معينة، أو التعامل مع المخازن المؤقتة للبيانات. هذا يسمح لك بالتعامل مع جزء من مساحة القرص كـ "ملف".

يمكنك بشكل مشابه أن تضع أنواع البيانات المجردة في طبقات. فإنه بإمكانك استخدام نوع البيانات المجردة لتزويد عمليات على مستوى هيكلة البيانات، (مثل الدفع أو السحب من المكدس). وإنه بإمكانك توليد مستوى أخر فوق ذلك المستوى، الذي يعمل في مستوى مشكلة العالم الحقيقي.

تعامل حتى مع العناصر الأكثر بساطة كأنواع بيانات مجردة ليس من الضرورة أن يكون لديك نوع بيانات هائلة، لتستطيع استخدام نوع البيانات المجردة. كما لاحظنا سابقا في قائمة الأمثلة، فإن واحدة من أنواع البيانات المجردة هي الضوء، الذي يدعم فقط عمليتين- تشغيل وإطفاء الضوء. من الممكن أن تعتقد أن عملية عزل العمليات البسيطة (تشغيل/اطفاء) في إجرائيات خاصة بها، ستكون هدراً للوقت، ولكن حتى العمليات البسيطة تستفيد من استخدام أنواع البيانات المجردة. حيث أن وضع الضوء وعملياته في نوع بيانات مجردة يجعل الشفرة ذاتية التوثيق بشكل أكبر، ويجعلها سهلة التغيير، ويحد من العواقب المحتملة من تغيير الإجرائيات `TurnLightOn()` و `TurnLightOn()`، ويقلل من عدد عناصر البيانات التي عليك أن تمر عبرها.

أشر إلى نوع البيانات المجردة بشكل مستقل عن الوسط الذي يخزن هذا النوع افترض أنه لديك جدول بمعدلات التأمين، كبير بما فيه الكفاية لتخزينه على قرص. من الممكن أنك ستميل إلى الإشارة إلى الجدول كـ "ملف معدلات التأمين"، وستقوم بتوليد إجرائيات الوصول مثل `RateFile.Read ()`. بقيامك بالإشارة إلى الجدول كملف، فإنك ستكشف أكثر مما تحتاج إليه من المعلومات حول البيانات. إذا قمت بتغيير البرنامج، بطريقة يصبح فيه الجدول في الذاكرة وليس على القرص، فإن الشفرة، التي تؤشر إلى الجدول كملف ستصبح غير صحيحة، ومربكة ومضللة. لهذا حاول أن تجعل أسماء الصفوف وإجرائيات الوصول مستقلة عن الطريقة التي يتم فيها تخزين البيانات، وتشير إلى نوع البيانات المجردة، مثل جدول معدلات التأمين. هذا سيعطي لصفوفك وإجرائيات الوصول أسماء مثل `rateTable.Read()` أو بشكل مبسط فقط `rates.Read()`.

التعامل مع حالات البيانات المتعددة باستخدام أنواع البيانات المجردة في البيئات غير غرضية التوجه

تزوّد اللغات غرضية التوجه دعم أتوماتيكي للتعامل مع الحالات المتعددة لأنواع البيانات المجردة. إذا عملت بشكل حصري في البيئات غرضية التوجه، ولم تتعامل أبداً مع تفاصيل التنفيذ للحالات المتعددة، فيمكنك الانتقال إلى القسم التالي "أنواع البيانات المجردة والصفوف".

وإذا عملت في بيئة غير غرضية التوجه مثل لغة البرمجة سي، فسيكون عليك بناء دعم للحالات المتعددة بشكل يدوي. هذا يعني، بشكل عام، تضمين خدمات في نوع البيانات المجردة لتوليد وحذف الحالات وتصميم خدمات أخرى لنوع البيانات المجردة، وبالتالي هذه الخدمات تستطيع أن تعمل مع حالات متعددة.

يمكن لنوع البيانات المجردة للخط تزويد الخدمات التالية، أصلاً:

```
currentFont.SetSize (sizeInPoints)
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace (faceName)
```

ولكن في البيئة غير غرضية التوجه، فإنه لا يمكن إسناد هذه التوابع لصف، وهذه التوابع ستبدو بالشكل:

```
SetCurrentFontSize (sizeInPoints)
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
SetCurrentFontTypeFace (faceName)
```

وإذا كنت ترغب بإضافة أكثر من نوع خط في نفس الوقت، فإنه ستحتاج إلى إضافة الخدمات التالية لتوليد وحذف حالات أنواع الخط، من الممكن بهذا الشكل:

```
CreateFont( fontId )
DeleteFont( fontId )
SetCurrentFont( fontId )
```

حيث تمّ إضافة " `fontId` " للتمكن من ملاحقة أنواع الخط المتعددة، عندما يتم توليدها واستخدامها. من أجل العمليات الاخرى، يمكنك الاختيار بين ثلاث طرق للتعامل مع واجهة نوع البيانات المجردة:

- الخيار الأول: التعريف الصـــريح للحالات في كل مرة ترغب فيها باســـتخدام خدمات نوع البيانات المجردة. في هذه الحالة، لن تكون بحاجة إلى اســـتخدام "current font"- بل ســـتمرر المعرّف "fontId" إلى كل إجرائية تعالج الخط. تُلاحق توابع الخط أية بيانات أســـاســـية، وشـــفرة الزبون عليها فقط أن تلاحق المعرّف "fontId". هذا يتطلب إضافة المعرّف "fontId" كوسيط في كل إجرائية تحرير الخط.
- الخيار الثاني: التزويد الصـــريح للبيانات المســـتخدمة من قبل خدمات نوع البيانات المجردة. في هذه الطريقة، عليك أن تصـــرّح عن البيانات التي يســـتخدمها نوع البيانات المجردة داخل كل إجرائية تســـتخدم خدمة نوع البيانات المجردة. بكلمات أخرى، تقوم بتولد نوع البيانات "Font"، الذي يمكن تمريره لكل من إجرائيات خدمة نوع البيانات المجردة. ثم عليك تشكيل إجرائيات خدمة نوع البيانات المجردة، بحيث يمكن لهذه الإجرائيات أن تســـتخدم نوع البيانات "Font"، الذي يتم تمريه لهم في كل مرة يتم فيها اســـتدعائهم. لا تحتاج شـــفرة الزبون إلى معرفة المعرّف fontID، لأن هذه الطريقة تلاحق

بنفسها بيانات النوع "Font". (بالرغم من أن البيانات متاحه بشكل مبا شر من نوع البيانات "Font"، إلا أنه يجب الوصــول إليها فقط عن طريق إجرائيات الخدمة لنوع البيانات المجردة. هذا يُدعى "ابقاء البنية مغلقة").

الميزة من هذه الطريقة، أنه لا يجب على إجرائيات خدمة نوع البيانات المجردة أن تبحث على معلومات بيانات الخط اســتناداً إلى معرِّف الخط "ID". وعيب هذه الطريقة، أنها تعرض بيانات الخط لبقية البرنامج، مما يزيد من احتمال أن شـفرة الزبون سـتسـتفيد من تفاصـيل التنفيذ، التي يجب أن تكون مخفية داخل نوع البيانات المجردة.

- الخيار الثالث: اسـتخدم حالات مضـمنة (مع عناية كبيرة). صـمم خدمة جديدة، التي يمكن اسـتدعائها لتشـكيل حالة خاصـة للخط من الحالة الحالية – مثل SetCurrentFont(fontId). إن ضـبط الخيار الحالي للخط سوف يجعل كل الخدمات الأخرى تستخدم الخط الحالي عندما يتم استدعاء هذه الخدمات. إذا كنت تســتخدم هذه الطريقة، فلن تكون هناك حاجة لاســتخدام المعرّف "fontId" كوســيط للخدمات الأخرى. بالنســبة للتطبيقات البســيطة، هذا يعتبر تبســيط لاســتخدام الحالات المتعددة. بالنســبة للتطبيقات المعقدة، هذا الاعتماد هو في نطاق النظام يعني أنه عليك أن تتبع الحالة الحالية للخط في الشـفرة التي تسـتخدم توابع تحرير. في هذا الخيار يميل التعقيد إلى الزيادة، ويوجد خيارات أفضـل للتطبيقات من أي حجم.

ستملك خيارات ضعيفة للتعامل مع الحالات instances المتعددة داخل نوع البيانات المجردة، ولكن خارجها، فهذا يلخص الخيارات إذا كنت تستخدم لغة غير غرضية التوجه.

أنواع البيانات المجردة والصفوف

تُشكل أنواع البيانات المجردة الأساس لمفهوم الصفوف. في لغات البرمجة التي تدعم الصفوف، يمكنك تنفيذ كل نوع بيانات مجردة كصف خاص بهذا النوع. تتضمن الصفوف عادةً مفاهيم أخرى كالوراثة وتعدد الأشكال (polymorphism). إن أحد الطرق للتفكير بالصف، باعتباره كنوع بيانات مجردة زائد الوراثة وتعدد الأشكال.

2.6 واجهات الصف الجيدة

الخطوة الأولى، ومن الممكن الأهم، لتشكيل صف عالي الجودة، هي بتشكيل واجهة جيدة. هذا يتضمن تشكيل مستوى تجريدي جيد للواجهة التي سيتم تنفيذها، والتأكد من إخفاء التفاصيل خلف التجريدية.

التجريد الجيد

كما هو موصوف في القسم 3.5 "صيغ التجريدات المتسقة"، التجريد هي عبارة عن القدرة على رؤية عملية معقدة في صيغة بسيطة. تقدم واجهة الصف تجريد للتنفيذ المختبئ خلف الواجهة. يجب على واجهة الصف أن تعرض الإجرائيات، التي تنتمي بشكل واضح إلى بعضها البعض.

لنفترض لديك صف يمثل عامل. هذا الصف عليه أن يحتوي بيانات تصف اسم العامل وعنوانه ورقم هاتفه وإلى أخره. ومن الممكن أن يعرض خدمات، كتهيئة واستخدام عامل. وإليك كيف يبدو هذا:

مثال بلغة البرمجة سي++ عن واجهة صف، تُقدم تجريدية جيدة [1]

```
class Employee {
public:
  // البواني و الهوادم العامة
  Employee();
 Employee(
 FullName name,
 String address,
 String workPhone,
 String homePhone,
 TaxId taxIdNumber,
 JobClassification jobClass
 );
 virtual ~Employee();
  // الإجرائيات العامة
 FullName GetName() const;
 String GetAddress() const;
 String GetWorkPhone() const;
 String GetHomePhone() const;
 TaxId GetTaxIdNumber() const;
 JobClassification GetJobClassification() const;
...
private:
...
 };
```

[1] **إشارة مرجعية**: تم تنسيق عينات الشفرة في هذا الكتاب، باستخدام اتفاقية ترميز، تؤكد على تشابه أساليب الكتابة في عدة لغات برمجة. لمزيد من التفاصيل حول اتفاقية الترميز هذه، (والمناقشات حول أساليب الترميز المتعددة)، انظر "اعتبارات البرمجة بلغات برمجة مختلطة" في القسم 4.11

من الممكن أن يحوي هذا الصف، داخلياً، على إجرائيات إضافية، وبيانات لدعم هذه الخدمات، ولكن لا يوجد حاجة لمستخدم الصف لمعرفة هذا. يعتبر تجرد واجهة الصف ممتاز، لأن كل إجرائية في الواجهة تعمل باتجاه هدف متناسق. يعتبر الصف الذي يمثل تجريدية ضعيفة، كمجموعة من التوابع المتفرقة. إليك مثال على ذلك:



مثال بلغة البرمجة سي++ عن واجهة صف، تُقدم تجريدية ضعيفة

```
class Program {
public:
...
  // الإجرائيات العامة
  void InitializeCommandStack();
 void PushCommand( Command command );
 Command PopCommand();
 void ShutdownCommandStack();
 void InitializeReportFormatting();
 void FormatReport( Report report );
 void PrintReport( Report report );
 void InitializeGlobalData();
 void ShutdownGlobalData();
...
private:
...
 };
```

افترض يوجد صف يحتوي إجرائيات تعمل مع مكدس الأوامر، وتقوم بتشكيل التقارير وطباعة التقارير وتهيئة البيانات العامة. من الصعب رؤية أية ترابط بين مكدس الأوامر وإجرائيات التقرير أو البيانات العامة. لا تمثل واجهة الصف هذه تجريدية متسقة، وبالتالي يملك الصف تماسك ضعيف. ويجب إعادة تنظيم الإجرائيات في صفوف أكثر تركيزاً، وعلى كل صف تأمين تجريدية أفضل في واجهته. لو كانت هذه الإجرائيات جزء من صف في برنامج، فإنه يمكن تنقيحها لكي تمثل تجريداً متناسقاً، مثل ذلك:

مثال بلغة البرمجة سي++ عن واجهة صف، تُقدم تجريدية أفضل

```
class Program {
public:
...
 // الإجرائيات العامة
 void InitializeUserInterface();
 void ShutDownUserInterface();
```

```
 void InitializeReports();
 void ShutDownReports();
...
private:
...
 };
```

تتضمن إعادة تنظيم هذه الواجهة نقل بعض الإجرائيات الأصلية إلى صفوف أخرى مناسبة أكثر، وتحويل بعض الإجرائيات إلى إجرائيات خاصة مستخدمة من قبل `InitializeUserInterface()` وإجرائيات أخرى.

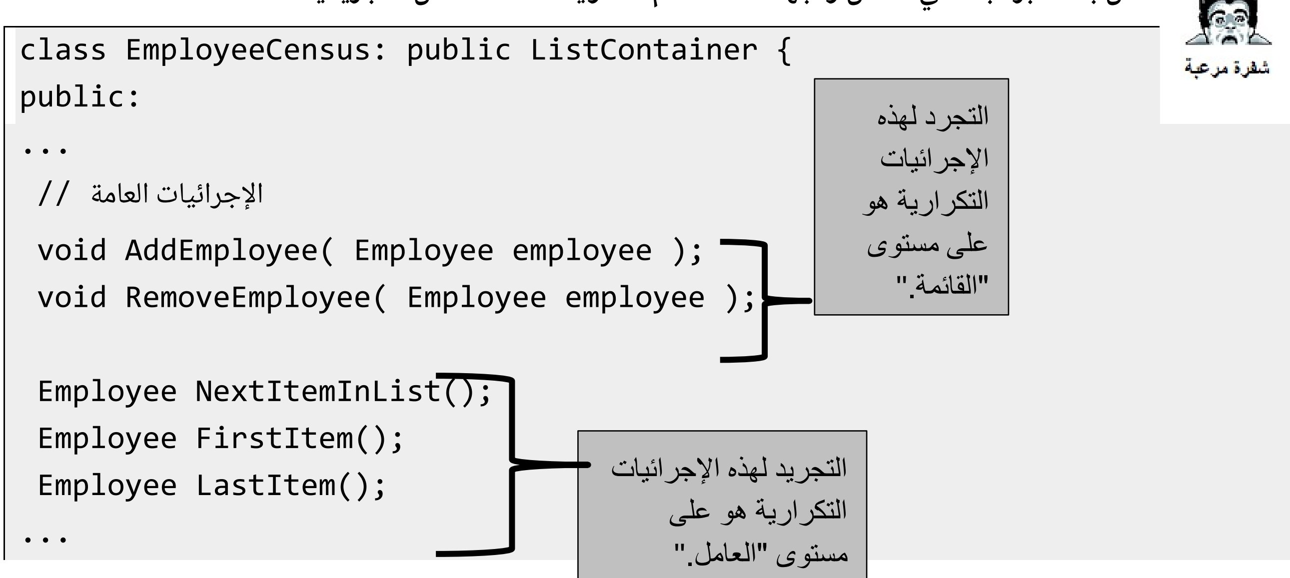
يستند هذا التطوير في تجريدية الصف على تجميع الإجرائيات العامة للصف، الموجود على واجهة الصف. وليس على الإجرائيات داخل الصف أن تقدم مستوى تجريدي جيد فقط لأن كامل الصف يقدم ذلك، بل لأنها تحتاج أيضاً لأن يتم تصميمها بطريقة تقدم فيها تجريدية جيدة. للحصول على إرشادات حول ذلك، راجع القسم 2.7"التصميم على مستوى الإجرائية".

يؤدي هذا السعي وراء توليد واجهات مجردة جيدة، إلى العديد من المبادئ التوجيهية لتوليد واجهات الصف:

قدّم مستوى ثابت من التجريدية في واجهة الصف في القسم 1.6 موصوفة طريقة جيدة للتفكير بالصف كآلية لتنفيذ أنواع البيانات المجردة. يجب على كل صف أن ينفذ نوع بيانات مجردة واحد فقط لا غير. إذا وجدت أن صف ينفذ أكثر من نوع بيانات مجردة، أو لم تستطيع تحديد ما النوع الذي ينفذه هذا الصف، هذا يعني قد حان الوقت لإعادة تنظيم الصف في واحد أو أكثر من أنواع البيانات المجردة المعرفة بشكل جيد. فيما يلي مثال عن صف يقدم واجهة غير متناسقة، بسبب أن مستوى التجريد غير موحّد:


مثال بلغة البرمجة سي++ عن واجهة صف، تُقدم مستويات مختلطة من التجريدية

شفرة مرعبة



```
class EmployeeCensus: public ListContainer {
public:
...
 // الإجرائيات العامة
 void AddEmployee( Employee employee );
 void RemoveEmployee( Employee employee );

 Employee NextItemInList();
 Employee FirstItem();
 Employee LastItem();
...
```

```
private:
...
 };
```

يحوي هذا الصف على نوعين للبيانات المجردة: "Employee" و "ListContainer". يظهر هذا النوع من التجريد المختلط، عندما يستخدم المبرمج صف حاوي أو صفوف مكتبة أخرى للتنفيذ، وفي نفس الوقت لا يخفي حقيقة استخدامه للصف من المكتبة. اسأل نفسك فيما إذا كان الصف الحاوي المستخدم يجب أن يكون جزء من التجريدية. عادة يجب أن تكون تفاصيل التنفيذ هذه مخفية عن بقية البرنامج، مثل هذا:

مثال بلغة البرمجة سي ++ عن واجهة صف، تُقدم مستويات متسقة من التجريدية

```
class EmployeeCensus {
public:
...
 // الإجرائيات العامة
 void AddEmployee( Employee employee );
 void RemoveEmployee( Employee employee );
 Employee NextEmployee();
 Employee FirstEmployee();
 Employee LastEmployee();
...
private:
 ListContainer m_EmployeeList;
...
 };
```


يجادل المبرمجون أن الوراثة من ListContainer هي مناسبة، لأنها تدعم تعدد الأشكال، مما يسمح ببحث خارجي أو بتابع تصنيف يأخذ كائن الـ ListContainer. تفشل هذه الحجة في الاختبار الرئيسي للوراثة، الذي هو "هل يستخدم الميراث فقط للعلاقات "is a" "هو"؟. تعني الوراثة من ListContainer أن EmployeeCensus "هو" ListContainer، وهذا من الواضح أنه غير صحيح. إذا كان التجريد من الكائن EmployeeCensus من الممكن أن يتم البحث فيه أو تصنيفه، فيجب إدراجه كجزء صريح ومتسق من واجهة الصف.

إذا كنت تفكر في الإجرائيات العامة للصف بمثابة قفل الهواء الذي يحافظ على المياه من الدخول إلى الغواصة، فالإجرائيات العامة غير المتسقة هي لوحات التسرب في الصف. من الممكن ألا تسمح لوحات التسرب بتسرب الماء بشكل كبير كما في حالة القفل الهوائي المفتوح، ولكن إذا أعطيتها وقت كافي، فإنها ستغرق القارب. في الحياة العملية، هذا ما يحدث عند خلط مستويات من التجريد. بالإضافة إلى تعديل البرنامج، فإن خلط

مستويات من التجريدية سوف يجعل البرنامج أصعب وأصعب للفهم، وهذا سوف يزداد تدريجياً حتى يصبح البرنامج غير قابل للصيانة.

نقطة مفتاحية

كُن متأكد من فهمك ماذا ينفذ تجريد الصف بعض الصفوف تكون متشابهة بشكلٍ كافٍ، مما يجبرك على أن تكون حذر في فهم أي تجريد يجب ان تمثله واجهة الصف. في أحد المرات عملت على برنامج، وكنت احتاج للسماح بتحرير المعلومات في تنسيق جدول. احتجنا وقتها إلى شبكة تحكم بسيطة، ولكن شبكة التحكم التي كانت متاحة لم تسمح لنا بتلوين خلايا إدخال البيانات، لذلك قررنا أن نستخدم عنصر تحكم جدولي (`spreadsheet control`) والذي يوفر تلك القدرة.

وكان عنصر التحكم الجدولي أكثر تعقيداً من شبكة التحكم، حيث احتوى على 150 إجرائية مقابل 15 إجرائية لشبكة التحكم. وبما أن هدفنا كان استخدام شبكة التحكم وليس عنصر التحكم الجدولي، فعيّنا مبرمج ليقوم بكتابة الصف الغلاف لإخفاء حقيقة أننا نستخدم عنصر التحكم كشبكة تحكم الجدولي. تطرق المبرمج قليلاً إلى النفقات العامة غير الضرورية والأمور الروتينية وذهب، وعاد بعد عدة أيام بالصف الغلاف، الذي عرض بأمانة كل الإجرائيات الـ 150 لعنصر التحكم.

لم يكن هذا هو المطلوب. بل احتجنا إلى واجهة لشبكة التحكم التي تخفي خلف الكواليس حقيقة استخدامنا لعنصر تحكم جدولي أكثر تعقيداً. كان على المبرمج أن يعرض فقط 15 إجرائية لشبكة التحكم، زائد الإجرائية السادسة عشر، التي تدعم تلوين الخلايا في الجدول. قدم لنا المبرمج بعرضه لـ 150 إجرائية، امكانية استخدام إجرائية عامة في حال أردنا تغيير التنفيذيات الرئيسية. قد فشل المبرمج في التغليف الذي كنا نبحث عنه، فضلاً عن الكثير من العمل غير الضروري الذي رتبه على نفسه.

بالاعتماد على الظروف الخاصة، فإن التجريد الصحيح من الممكن أن يكون إما شبكة التحكم أو عنصر التحكم الجدولّي. عندما يكون عليك الاختيار بين اثنين من التجريدات المتماثلة، تأكد من اختيار التجريد الصحيح.

قدّم الخدمات على شكل أزواج، أي مع أضدادهم معظم العمليات لها عمليات مقابلة وعمليات مساوية وعمليات معاكسة. مثلاً إذا كان لديك عملية تقوم بتشغيل الضوء، فإنك على الأغلب ستحتاج إلى عملية لإطفاء الضوء. ومثلاً إذا كان لديك عملية تقوم بإضافة عنصر إلى قائمة، فإنك على الأغلب ستحتاج إلى عملية لحذف عنصر من القائمة. ومثلاً إذا كان لديك عملية تقوم بتفعيل خيار في قائمة خيارات، فإنك على الأغلب ستحتاج إلى عملية لتعطيل خيار في قائمة الخيارات. لهذا عندما تقوم بتصميم صف، عليك أن تفحص فيما إذا هناك حاجة لمتمم لكل إجرائية عامة في الصف. قُم دائما بتشكيل الإجرائية المعاكسة، بل تأكد دائما من وجود حاجة لها أو لا.

انقل المعلومات غير المتصلة إلى صف آخر في بعض الحالات ستجد أنّ نصف إجرائيات الصف تعمل مع نصف بيانات الصف، والنصف الأخر من الإجرائيات تعمل مع النصف الآخر من بيانات الصف. في هذه الحالة عليك حقاً كسر الصف إلى صفين.

اجعل الواجهات تصويرية بدلا من دلالية، قدر الإمكان تحوي كل واجهة على قسم تصويري وقسم دَلَالِيّ. يحتوي القسم التصويري على أنواع البيانات وصفات أخرى للواجهة، التي يمكن فرضها من قبل مترجم الشفرة (compiler). يحتوي القسم الدَلَالِيّ على افتراضات حول كيفية استخدام الواجهة، هذه الافتراضات لا يمكن تنفيذها من قبل المترجم. تحتوي الواجهة الدلالية على اعتبارات معينة مثل يجب استدعاء "الإجرائية أ" قبل "الإجرائية ب" أو ستتعطل "الإجرائية أ" إذا لم يتم تهيئة "عنصرالبيانات1" قبل تمريره "للإجرائية أ". يجب أن يتم توثيق الواجهة الدلالية في تعليقات، ولكن حاول أن تحافظ على اعتماد الواجهات بشكل قليل على التوثيق. يعد أي جانب من جوانب الواجهة، التي لا يمكن تنفيذها من قبل المترجم، هو جانب يساء استخدامه على الأرجح. أبحث عن طرق تحويل عناصر الواجهة الدلالية إلى عناصر واجهة تصويرية باستخدام تقنية التأكيد "Asserts" أو باستخدام تقنيات أخرى.

احذر من تآكل تجريدية الواجهة تحت التعديل[1] عندما يتم تعديل الصف وتمديده، من الممكن أن تكتشف أنه هناك حاجة إلى وظائفية جديدة، لا تناسب تماماً واجهة الصف الأصلي، ولكن يبدو أنه من الصعب جداً تنفيذها بأي طريقة أخرى. على سبيل المثال، في الصف Employee، من الممكن أن تجد أن الصف يتطور ليبدو بهذا الشكل:



مثال بلغة البرمجة سي++ عن واجهة صف، يتأكل تحت الصيانة

```
class Employee {
public:
...
 // الإجرائيات العامة
 FullName GetName() const;
 Address GetAddress() const;
 PhoneNumber GetWorkPhone() const;
...
 bool IsJobClassificationValid( JobClassification jobClass );
 bool IsZipCodeValid( Address address );
 bool IsPhoneNumberValid( PhoneNumber phoneNumber );
 SqlQuery GetQueryToCreateNewEmployee() const;
```

[1] **إشارة مرجعية**: لمزيد من الاقتراحات حول كيفية الحفاظ على جودة الشفرة عندما يتم تعديله، انظر الفصل ٢٤" إعادة التصنيع".

```
 SqlQuery GetQueryToModifyEmployee() const;
 SqlQuery GetQueryToRetrieveEmployee() const;
...
private:
...
};
```

ما بدى كتجريد نظيف في عينة الشفرة السابقة قد تطوّر إلى خليط من التوابع التي ترتبط فيما بينها بطريقة حرّه. لا يوجد أي رابط منطقي بين العمال والإجرائيات التي تفحص الأرقام البَرِيدِيّة أو أرقام الهاتف أو تصنيفات الأعمال. إن الإجرائيات التي تعرض تفاصيل الاستعلام SQL هي عند مستوى تجريدية أقل بكثير من الصف Employee، وتقوم هذه الإجرائيات بكسر تجريد الصف Employee.

لا تضف عناصر عامة لا ترتبط مع تجريد الواجهة. في كل مرة تقوم بإضافة إجرائية إلى واجهة الصف اسأل نفسك السؤال التالي "هل هذه الإجرائية تتناسق مع التجريد المقدّم من الواجهة الموجودة؟"، إذا كان الجواب لا، فيجب إيجاد طريقة أخرى للتعديل والمحافظة على التكاملية مع التجريدية.

خذ بعين الاعتبار التجريد والترابط معاً إن فكرة التجريد وفكرة الترابط مرتبطين بشكل كبير. إن واجهة الصف التي تقدّم تجريد جيد تملك في الغالب ترابط قوي. تميل الصفوف ذات الترابط القوي إلى تقديم تجريدات جيدة، على الرغم من أن هذه العلاقة ليست قوية.

لقد وجدت أن التركيز على التجريد المقدم من قبل واجهة الصف يميل إلى توفير المزيد من الفهم في تصميم الصف، أكثر من التركيز على ترابط الصف. إذا كنت ترى أن الصف يملك ترابط ضعيف، ولست متأكدا كيف تستطيع إصلاح هذا، فاسأل نفسك فيما إذا كان يقدم الصف بدلاً من ذلك تجريد ثابت.

التغليف الجيد

كما تم مناقشته في القسم 5-3، إن التغليف مفهوم أقوى من التجريد[1]. يساعد التجريد على إدارة التعقيد، عن طريق تزويد نماذج، تسمح لك بتجاهل تفاصيل التنفيذ. أما التغليف فهو المفصل الذي يمنعك من النظر في التفاصيل حتى إذا كنت تريد ذلك.

هذين المفهومين مرتبطين، لأنه بدون التغليف، فإن التجريد يميل إلى الفشل. بناءً على خبرتي، فإما أن تملك التجريد والتغليف مع بعض أو لا تملك ولا واحدة، لا يوجد حل وسط.

[1] **إشارة مرجعية**: للمزيد حول التغليف، انظر "تغليف تفاصيل التنفيذ" في القسم 5-3.

قلل من إمكانية الوصول إلى الصفوف والعناصر[1] إن الحدّ من أمكانية الوصول هي واحدة من القواعد المتعددة التي تعزز التغليف. إذا كنت تتسأل فيما إذا كان على الصف أن يكون عام أو خاص أو محمي، فإن أحد مدارس التفكير تقول أنّه يجب عليك أن تفضل المستوى الأكثر صرامة من الخصوصية (مايرز 1998، بلوش 2001) (Meyers 1998, Bloch 2001). أعتقد أن هذا المبدأ التوجيهي لا بأس به، ولكن المبدأ الأهم هو "اختر الأفضل الذي يحافظ على سلامة تجريد الواجهة". فإذا كان عرض الإجرائية متناغم مع التجريد، فإنه ربما أحسن إظهار ذلك. ولكن إذا لم تكن متأكد فإن إخفاء الكثير عموماً أفضل من إخفاء القليل.

لا تعرض بيانات العناصر بشكل عام يمثل عرض بيانات العناصر انتهاكاً للتغليف، ويحدّ من تحكمك بالتجريد. كما يشير آرثر ريل (Arthur Riel)، إن الصف Point الذي يعرض:

```
float x;
float y;
float z;
```

يُمثل انتهاكاً للتغليف، لأن شفرة الزبون مفتوحة للتلاعب ببيانات الصف Point، الذي ليس عليه بالضرورة أن يعرف متى تغيرت قيمه (ريل 1996) (Riel 1996)، أما الصف Point الذي يعرض:

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

يحافظ على تغليف أفضل. حيث هنا لا تملك أدنى فكرة فيما إذا كان التنفيذ الأساسي لـ x, y, z كعدد حقيقي، أو فيما إذا كان الصف Point يخزنهم كأعداد حقيقية مضاعفة الدقة doubles، ومن ثم يحولهم إلى أعداد حقيقية من النوع floats، أو فيما إذا كان الصف Point يخزنهم على القمر ويسترجعهم من قمر صناعي في الفضاء الخارجي.

تجنب وضع تفاصيل التنفيذ الخاص داخل واجهة الصف بوجود تغليف حقيقي، لا يمكن أبداً للمبرمجين رؤية تفاصيل التنفيذ. حيث ستكون هذه التفاصيل مخفية بشكل مجازي أو حرفي. في بعض لغات البرمجة مثل سي++، تتطلب بنية اللغة من المبرمجين أن يكشفوا تفاصيل التنفيذ في واجهة الصف. لدينا هنا مثال:

مثال بلغة البرمجة سي++ عن إظهار تفاصيل تنفيذ الصف

[1] العامل الوحيد الأكثر أهمية، الذي يميز الوحدة المصممة تصميما جيدا عن الوحدة المصممة تصميماً ضعيفاً، هو درجة إخفاء الوحدة لبياناتها الداخلية وتفاصيل التنفيذ الأخرى عن الوحدات الأخرى. - جوشوا بلوخ (Joshua Bloch)

```
class Employee {
public:
...
 Employee(
 FullName name,
 String address,
 String workPhone,
 String homePhone,
 TaxId taxIdNumber,
 JobClassification jobClass
 );
...
 FullName GetName() const;
 String GetAddress() const;
...
private:
 String m_Name;
 String m_Address;
 int m_jobClass;
...
};
```

هنا تفاصيل التنفيذ المعروضة.

يبدو تضمين التصريحات الخاصة في الملف الأساسي للصف انتهاكاً صغيراً، ولكنه يشجع المبرمجين الآخرين لتفحص تفاصيل التنفيذ. في هذه الحالة، من الممكن أن تميل شفرة الزبون إلى استخدام النوع Address للعناوين، ولكن الملف الرئيسي يظهر تفاصيل التنفيذ، بما فيها أن العناوين تخزن كنوع Strings.

يصف سكوت مايرز (Scott Meyers) طريقة مشتركة لمعالجة هذه المسألة في البند 34 من " سي++ الفعّال- الإصدار الثاني". (مايرز 1998).(Item 34 of Effective سي++, 2d ed. (Meyers 1998).) يمكنك فصل واجهة الصف عن تنفيذ الصف داخل تعريف الصف، وتضمين مؤشر لتنفيذ الصف ولكن لا تضمن أي تفاصيل تنفيذ أخرى.

مثال بلغة البرمجة سي++ عن إخفاء تفاصيل تنفيذ الصف

```
class Employee {
public:
...
 Employee(... );
...
```

```
 FullName GetName() const;
 String GetAddress() const;
...
private:
 EmployeeImplementation *m_implementation;
};
```

هنا تفاصيل التنفيذ مختبئة خلف المؤشر.

الآن تستطيع أن تضع تفاصيل التنفيذ داخل الصف EmployeeImplementation، الذي يجب أن يكون مرئياً فقط للصف Employee، وليس للشفرة التي تستخدم الصف Employee.

إذا كنت قد قمت للتو بكتابة العديد من الشفرات التي لا تستخدم هذه الطريقة في مشروعك، من الممكن أنك ستقرر، أنه لا يستحق الجهد تحويل جبل من التعليمات البرمجية الموجودة في الشفرة لهذه الطريقة. ولكن عندما تقرأ الشفرة التي تعرض تفاصيل التنفيذ، سوف تقاوم الرغبة بالغوص في القسم الخاص من واجهة الصف والبحث عن ألغاز التنفيذ.

لا تضع افتراضات حول مستخدمي الصف يجب أن يكون الصف مصمم ومنفذ وفق العقد الذي تنطوي عليه واجهة الصف. لا يجب وضع افتراضات حول الكيفية التي ستُستخدم بها الواجهة أو لن تُستخدم، بخلاف ما هو موثق في الواجهة. إن تعليقات مثل التالية، هي مؤشر على أن الصف أكثر وعياً حول مستخدميه مما ينبغي أن يكون:

```
-- initialize x, y, and z to 1.0 because DerivedClass blows
-- up if they're initialized to 0.0
```

تجنب الصفوف الصديقة في عدد قليل من الظروف مثل نمط الحالة، يمكن استخدام الصفوف الصديقة بطريقة منظمة، من شأنها أن تسهم في إدارة التعقيد (غاما إت آل. 1995) (Gamma et al. 1995). ولكن، بشكل عام، تنتهك الصفوف الصديقة مبدأ التغليف. حيث توسع الصفوف الصديقة كمية الشفرة، الذي عليك التفكير فيه في أي وقت، وبالتالي زيادة التعقيد.

لا تضع إجرائية في واجهة عامة، فقط لأنها تستخدم الإجرائيات العامة لا تعد حقيقة استخدام إجرائية فقط لإجرائيات عامة، اعتبار أساسي. بدلاً من ذلك، اسأل فيما إذا كان عرض الإجرائية سيكون متناغم مع التجريد المقدم من الواجهة.

فضّل وقت القراءة على وقت الكتابة يتم قراءة الشفرة مرات عدة أكثر من كتابتها، حتى ولو كان هذا أثناء التطوير الأولي للشفرة. إن تفضيل التقنية التي تسرع وقت الكتابة على حساب وقت القراءة، هي تدبير خاطئ. ينطبق هذا بوجه خاص على إنشاء واجهات الصفوف. حتى لو أن الإجرائية لا تناسب تجريدية الواجهة، ففي بعض الأوقات إنه من المغري إضافة الإجرائية إلى الواجهة لتناسب زبون محدد للصف، الذي تعمل عليه في هذا

الوقت. ولكن إضافة هذه الإجرائية هي الخطوة الأولى على منحدر زلق للخطأ، ومن الأفضل عدم اتخاذ حتى الخطوة الأولى.

كن حذرا جدا من الانتهاكات الدلالية للتغليف[1] في أحد المرات اعتقدت أنه إذا تمكنت من التخلص من أخطاء تركيب الجملة البرمجية، فسأكون حراً. ولكن سرعان ما اكتشفت أن تعلم كيفية تجنب أخطاء تركيب الجملة قد اشترى لي فقط تذكرة لمسرح جديد من أخطاء التشفير، ومعظمها كانت أكثر صعوبة في تشخيصها وتصحيحها من أخطاء تركيب الجملة البرمجية.

إن صعوبة التغليف الدلالي بالمقارنة مع التغليف النحوي متشابهة. نحوياً، من السهل نسبياً تجنب حشر أنفك في الأعمال الداخلية لصف أخر، فقط عن طريق التصريح عن الإجرائيات الداخلية للصف والبيانات من النوع الخاص `private`. أما التغليف الدلالي هو مسألة أخرى تماما. هنا بعض الأمثلة عن الكيفية التي يستطيع فيها مستخدم الصف كسر التغليف بشكل دلالي:

- لا تقوم باستدعاء إجرائية الصف A InitializeOperations()، لأنك تعرف أن إجرائية الصف A () PerformFirstOperation تستدعيه بشكل اتوماتيكي.
- لا تستدعي إجرائية الاتصال بقاعدة البيانات database.Connect() قبل استدعاء التابع employee.Retrieve(database)، لأنك تعرف أن التابع employee.Retrieve() سيتصل مع قاعدة البيانات إذا لم يكن هذه الاتصال للتو موجود.
- لا تستدعي إجرائية الصف A Terminate()، لأنك تعرف أن إجرائية الصف A PerformFinalOperation() قد قامت باستدعائها للتو.
- استخدم مؤشر أو مرجع للكائن ObjectB، الذي تم انشائه من قبل الكائن ObjectA، حتى لو خرج الكائن ObjectA خارج النطاق، لأنك تعرف أن الكائن ObjectA يحتفظ بالكائن ObjectB في وحدة تخزين ساكنة، وبالتالي فإن الكائن ObjectB دائماً سيكون متاح.
- استخدم ثابت الصف B MAXIMUM_ELEMENTS بدلاً من استخدام الثابت ClassA.MAXIMUM_ELEMENTS، لأن كلاهما متساويين بالقيمة.



إن المشكلة في كل واحده من هذه الأمثلة أنها تقوم بجعل شفرة الزبون مستقلة ليس عن واجهة الصف العامة بل عن تنفيذه الخاص. في كل مرة تجد فيها نفسك تنظر إلى تنفذ الصف لفهم كيف يتم استخدم الصف، فأنت لا تبرمج الواجهة، بل تقوم ببرمجة التنفيذ من خلال الواجهة. وإذا كنت تقوم بالبرمجة من خلال الواجهة، هذا يعني أن التغليف انكسر، وإذا التغليف بدأ بالانكسار فهذا يعني أن التجريد سيلحقه عن قريب.

[1] أنه ليس بتجريد إذا كان عليك النظر في التنفيذات الأساسية لفهم ماذا يحدث - ب. ج. بلوجر (P. J. Plauger)

إذا لم تكن قادر على فهم كيفية عمل الصف فقط من خلال التوثيق الموجود في واجهته، فإن ردّة الفعل الصحيحة ليس بسحب شفرة المصدر والنظر في التنفيذ. هذه مبادرة جيدة ولكنها حكم سيء. ردة الفعل الصحيحة هي الاتصال بكاتب الصف وأن تقول له "لا أستطيع فهم كيف يعمل هذا الصف". أما ردة الفعل الصحيحة من جانب المؤلف، هي ليس بالرد على سؤالك وجهاً لوجه. بل هي بفحص ملف واجهة الصف، وتعديل التوثيق الموجود فيه، ومن ثم فحص الملف مرة أخرى، وبعدها القول " انظر إذا كنت تستطيع أن تفهم الآن كيف يعمل الصف ". تريد هذا الحوار أن يحدث في شفرة الواجهة نفسه، وبالتالي سيتم حفظه للمبرمجين المستقبليين. لا تريد أن يحدث هذا الحوار فقط في دماغك، الذي سيؤدي إلى التبعيات الدلالية في شفرة الزبون الذي يستخدم الصف. وأنت لا تريد أن يحدث هذا الحوار بين شخصين، عندها فقط تستفيد أنت من شفرتك وليس أي شخص آخر.

انتبه إلى الاقتران الضيق يُشير "الاقتران" إلى أي حد الاتصال بين صفين ضيق. بشكل عام، كلما كان الاتصال فضفاض، كلما كان ذلك أفضل. تنشأ العديد من المبادئ التوجيهية حول هذا المفهوم:

- التقليل من الوصول إلى الصفوف والعناصر.
- تجنب صداقة الصفوف، لأنهم يقترنون بإحكام.
- اجعل البيانات خاصة بدلاً من محميّة في الصف القاعدة، وذلك لجعل الصفوف المشتقة أقل ارتباطاً بالصف القاعدة.
- تجنب عرض بيانات العناصر في واجهة عامة لصف.
- كن حذرا من الانتهاكات الدلالية للتغليف.
- لاحظ "قانون ديميتر" (Law of Demeter) (المناقش في القسم 6.3 من هذا الفصل).

إن الاقتران يرتبط مع التجريد والتغليف. يحدث الاقتران الضعيف عندما تكون التجريد ضعيف أو التغليف مكسور. إذا كان يعرض الصف مجموعة غير مكتملة من الخدمات، فمن الممكن للإجرائيات الأخرى أن تحتاج إلى كتابة أو قراءة بيناتها الداخلية بشكل مباشر. يفتح هذا الصف، مما يجعل منه صندوق زجاجي بدلاً من صندوق أسود، ويزيل عملياً تغليف الصف.

3.6 مشاكل التصميم والتنفيذ

يقطع تعريف واجهات الصف الجيد شوطاً طويلاً نحو إنشاء برنامج عالي الجودة. ويعد أيضاً تصميم الصف الداخلي والتنفيذ هاماً. يناقش هذا القسم المسائل المتعلقة بالاحتواء، والوراثة، وتوابع العناصر، والبيانات، واقتران الصف، والبواني، وكائنات القيمة مقابل المرجع.

الاحتواء (علاقات "يملك")



الاحتواء هو عبارة عن الفكرة البسيطة لاحتواء عنصر بيانات أساسي أو كائن. في المراجع مكتوب أكثر حول الوراثة بالمقارنة مع الاحتواء، ولكن هذا لأن الوراثة أكثر صعوبة وعرضه للخطأ، وليس لأنها الأفضل. إن الاحتواء هو تقنية العمل الأساسية للبرمجة غرضية التوجه.

نفذ علاقة "يملك" من خلال الاحتواء. أحد الطرق لفهم الاحتواء هو علاقة "يملك". على سبيل المثال، العامل "يملك" اسم، و "يملك" رقم هاتف، و "يملك" رقم ضريبي، وإلى آخره. ويمكنك عادةً تحقيق هذا عن طريق جعل الاسم ورقم الهاتف والرقم الضريبي عناصر بيانات للصف Employee.

نفّذ علاقة "يملك"، عن طريق الوراثة الخاصة كحل أخير في بعض الحالات، من الممكن أن تجد أنه لا يمكن إنجاز الاحتواء من خلال جعل كائن ما عنصر في كائن أخر. في هذه الحالة، يقترح بعض الخبراء الوراثة الخاصة من الكائن المحتوى (مايرز 1998، سوتر 2000) (Meyers 1998, Sutter 2000). الشيء الأساسي الذي سوف تفعله هو إعداد الصف الحاوي للوصول الى توابع العنصر المحمي وبيانات العنصر المحمي للصف المحتوى. في الممارسة العملية، هذه الطريقة تولّد علاقة بشكل مفرط مع الصف السلف وتنتهك مبدأ التغليف. يجب حل أخطاء التصميم بطريقة ما بدلاً من استخدام الوراثة الخاصة.

كن حذر من الصفوف، التي تحتوي على أكثر من حوالي سبعة عناصر بيانات تم العثور على أن العدد"7 ± 2"، هو عدد العناصر المنفصلة التي يمكن لشخص أن يتذكرها أثناء تنفيذ مهمه أخرى (ميلر 1956) (Miller 1956). إذا كان الصف يحوي على أكثر من سبع عناصر بيانات، خُذ بعين الاعتبار فيما إذا كان يجب تفكيك الصف إلى عدة صفوف أصغر (ريل 1996) (Riel 1996). من الممكن أن تخطأ باتجاه النهاية العالية 7±2، إذا كانت عناصر البيانات هي عناصر بيانات أساسية مثل الأعداد الصحيحة والسلاسل المحرفية، وباتجاه النهاية الدنيا 7±2 إذا كانت عناصر البيانات هي كائنات معقدة.

الوراثة (علاقات "هو")

الوراثة هي فكرة أن صف ما هو تخصيص لصف آخر. الغرض من الوراثة هو تشكيل شفرة بسيطة، عن طريق تعريف صف قاعدة يحدد العناصر الأساسية لاثنين أو أكثر من الصفوف المشتقة. العناصر المشتركة من الممكن أن تكون عبارة عن واجهات الإجرائية، أو التنفيذيات، أو عناصر البيانات، أو أنواع معطيات. تساعد الوراثة في تجنب الحاجة إلى تكرار الشفرة والبيانات في مواقع متعددة، عن طريق تركيز الشفرة داخل صف القاعدة.

عندما تريد أن تستخدم الوراثة، فعليك أن تتخذ عدة قرارات:

- من أجل كل إجرائية، هل ستكون الإجرائية مرئية بالنسبة للصفوف المشتقة؟ هل ستملك هذه الإجرائية تنفيذ افتراضي؟ هل من الممكن إعادة كتابة هذا التنفيذ الافتراضي؟
- من أجل كل عنصر من عناصر البيانات (المتغيرات، الثوابت، التعددات، وإلى أخره)، هل سيكون هذا العنصر مرئي بالنسبة للصفوف المشتقة؟

الأقسام الفرعيّة التالية سوف تقوم بشرح التفاصيل الدقيقة لهذه القرارات:

نفذ علاقة "هو" من خلال الوراثة العامة[1] عندما يقرر مبرمج أن يشكل صف جديد عن طريق الوراثة من صف موجود، هذا المبرمج يقول: الصف الجديد "هو" نسخة أكثر تخصصية من الصف القديم. ويحدّد صف القاعدة التوقعات بشأن كيفية تشغيل الصف المشتق، ويفرض قيوداً على كيفية عمل الصف المشتق (مايرز 1998) (Meyers 1998).

إذا لم يقوم الصف المشتق بالتقيد تماماً بعقد الواجهة المعرف من قبل الصف القاعدة، فإن الوراثة ليست بتقنية التنفيذ الصحيحة في هذه الحالة. عندها عليك إما استخدام الاحتواء أو القيام بتغييرات للتسلسل الهرمي للوراثة.

صمم ووثق الوراثة أو احظرها تُضيف الوراثة التعقيد إلى البرنامج، وبهذا فإنها تقنية خطيرة. وكما يقول أستاذ الجافا جوشوا بلوخ (Joshua Bloch)، " صمم ووثق الوراثة أو احظرها". إذا كان الصف مصمم بحيث لا يرثه صف آخر، فاجعل عناصره غير ظاهرية non-virtual في سي++، نهائية في جافا، أو بدون أسبقية في مايكروسوفت فيجوال بيسك، وبهذا لا يمكن أن ترث منه.

التزم بمبدأ ليسكوف للاستبدال (Liskov Substitution Principle) - LSP. في أحد المقالات عن البرمجة غرضيّة التوجه، ناقش باربرا ليسكوف (Barbara Liskov)، بأنه ليس عليك أن ترث من صف القاعدة إلا إذا كان الصف المشتق بشكل حقيقي "هو" نسخة أكثر تخصصيّة من الصف القاعدة (ليسكوف -1988) (Liskov 1988). لخّص أندي هانت و ديف توماس مبدأ LSP (Andy Hunt and Dave Thomas)، بالشكل التالي: "يجب على الصفوف الفرعية أن تكون مفيدة من خلال واجهة الصف القاعدة من دون أن يعلم المستخدم الاختلاف فيما بينهم." (هانت و توماس 2000) (Hunt and Thomas 2000).

بكلمات أخرى، كل الإجرائيات المعرّفة في الصف القاعدة يجب أن تعني نفس الشيء، عندما يتم استخدامها في كل صف مشتق.

---

[1] القاعدة الأساسية الهامة في البرمجة غرضية التوجه في سي++ هي: الوراثة العامة تعني "هو". ضع هذه القاعدة في الذاكرة. – سكوت مايرز (Scott Meyers)

إذا كان لديك صف قاعدة من Account وصفوف مشتقة من CheckingAccount، SavingsAccount، وAutoLoanAccount، فيجب على المبرمج أن يكون قادراً على استخدام أي من إجرائيات الصف القاعدة Account وأي من أنواعه المشتقة بدون القلق حول أي نوع فرعي يُمثل الكائن من الصف Account.

إذا كان البرنامج مكتوب وفق مبدأ LSP، فالوراثة أداة فعاله لتقليل التعقيد، لأنه عندها يستطيع المبرمج التركيز على الخواص العامّة لكائن ما بدون القلق حول التفاصيل. إذا كان على المبرمج أن يفكر بشكل دائم حول الاختلافات الدلالية في تنفيذات الصفوف المشتقة، عندها يزداد التعقيد بدلاً من أن ينقص. افترض أن لدى المبرمج الفكرة التالية: "إذا استدعيت الإجرائية InterestRate() في CheckingAccount أو SavingsAccount، فإنّها تعيد الفائدة التي يدفعها البنك، ولكن إذا قمت باستدعاء الإجرائية InterestRate() على AutoLoanAccount فيجب عليّ تغيير العلامة لأنها تعيد الفائدة التي يدفعها المستهلك إلى البنك." بالاستناد إلى المبدأ LSP، فإنه لا يجب على الصف AutoLoanAccount أن يرث صف القاعدة Account، لأن دلالات الإجرائية InterestRate( ) ليست نفس دلالات الإجرائية InterestRate() للصف القاعدة.

كُن متأكد من وراثة فقط لما ترغب بأن ترث يمكن للصف المشتق أن يرث واجهات إجرائية العنصر، أو التنفيذيات، أو كلاهما. يُظهر الجدول 6-1 الاختلافات في كيفية تنفيذ الإجرائيات وكيفية إعادة تعينها.

الجدول 6-1 الاختلافات في الإجرائيات الموروثة:

	قابلية إعادة التعيين	عدم قابلية إعادة التعيين
التنفيذ: افتراضي	إجرائية قابلة لإعادة التعيين	إجرائية غير قابلة لإعادة التعيين
التنفيذ: غير افتراضي	إجرائية مجرّدة قابلة لإعادة التعيين	غير مستخدمة (لا معنى من ترك إجرائية غير معرفة وغير مسموح لها بإعادة التعيين)

كما هو موضّح في الجدول، تأتي الإجرائيات الموروثة في ثلاث أنواع أساسية:

- إجرائية مجرّدة قابلة لإعادة التعيين، وهذا يعني أن الصف المشتق يرث واجهة الإجرائية ولكن لا يرث تنفيذها.
- إجرائية قابلة لإعادة التعيين، هذا يعني أن الصف المشتق يرث واجهة الإجرائية والتنفيذ الافتراضي، ولكن مسموح له فقط بإعادة تعيين التنفيذ الافتراضي.

- إجرائية غير قابلة لإعادة التعيين، هذا يعني أن الصف المشتق يمكن أن يرث واجهة الإجرائية والتنفيذ الافتراضي، ولكن غير مسموح له بإعادة تعيين تنفيذ الإجرائية.

عندما تريد تنفيذ صف جديد من خلال الوراثة، فكّر بأي نوع وراثة ترغب من أجل كل إجرائية. انتبه لوراثة التنفيذ، فقط لأنك ترغب بوراثة الواجهة، وانتبه لوراثة الواجهة، فقط لأنك ترغب بوراثة التنفيذ. إذا كنت ترغب باستخدام تنفيذ الصف، وليس واجهته، فاستخدم الاحتواء بدلاً من استخدام الوراثة.

لا تقوم "بإعادة تعيين" تابع غير قابل لإعادة التعيين تسمح كل من لغتي البرمجة جافا وسي++ نوعاً ما، للمبرمج لإعادة تعيين إجرائية غير قابلة لإعادة التعيين. إذا كان التابع خاص في صف القاعدة، فيستطيع الصف المشتق تشكيل تابع بنفس الاسم. عندها بالنسبة للمبرمج، فإن قراءة الشفرة في الصف المشتق، مثل هذا التابع من الممكن أن يؤدي إلى إرباك، لأنه يبدو مثل تعدد الأشكال، ولكنه بالحقيقة هو فقط تابع له نفس الاسم. لتوضيح هذه الفكرة بطريقة أخرى "لا تُعيد استخدام أسماء إجرائيات صف القاعدة غير القابلة لإعادة التعيين في الصفوف المشتقة ".

حرّك الواجهات والبيانات والسلوكيات المشتركة على أعلى مستوى ممكن في الشجرة الهرمية للوراثة كلما حركت الواجهات والبيانات والسلوك أعلى، كلما كان أسهل للصفوف المشتقة استخدامهم. ما هو الارتفاع الذي يعد عالياً جداً؟ دع التجريد يكون دليلك. إذا وجدت أن تحريك الإجرائية للأعلى سوف يكسر تجريد الكائن العالي، فعندها لا تقوم بهذا التحريك.

اشتبه بالصفوف، التي لها فقط حالة واحدة حيث يشير وجود حالة واحدة إلى أن التصميم يخلط الكائنات مع الصفوف. خُذ بعين الاعتبار إمكانية تشكيل كائن بدلاً من صف جديد. هل من الممكن أن يكون اختلاف الصف المشتق ممثل في البيانات بدلا من صف جديد؟ إنّ نمط الوليد المفرد هو استثناء واحد من هذا المبدأ التوجيهي.

اشتبه بالصفوف الأساسية، التي لها فقط صف مشتق واحد عندما أرى صف أساسي له فقط صف مشتق واحد، افترض أن المبرمج قد صممه "للمستقبل"-حاول التنبؤ باحتياجات المستقبل، ويقف هنا سؤال ما هي تلك الاحتياجات المستقبلية. إن أفضل طريقة للاستعداد للعمل المستقبلي ليست بتشكيل طبقات إضافية من الصفوف الأساسية، والتي "من الممكن أن يحتاجها أحدهم"؛ بل هي بجعل العمل الحالي واضحاً، ودقيقاً، وبسيطاً قدر الإمكان. وهذا يعني عدم تشكيل أية هياكل وراثة إضافية، إلا إذا كان هذا ضرورياً للغاية.

اشتبه بالصفوف التي تعيد تعيين إجرائية ما ولا تفعل شيئاً داخل الإجرائية المشتقة فهذا يُشير إلى خطأ في التصميم لصف القاعدة. على سبيل المثال، افترض أنك تملك صف `Cat` و إجرائية `Scratch()`، وافترض

أنك أخيراً وجدت أن بعض القطط منزوعة المخالب ولا يمكنها أن تخدش. من الممكن أن تميل إلى تشكيل صف مشتق من الصف Cat بالاسم ScratchlessCat، وتُعيد تعيين الإجرائية ()Scratch لكي لا تفعل شيئاً. تعرض هذه الطريقة عدة مشاكل:

- إنها تنتهك التجريد (عقد الواجهة) المقدم في الصف Cat، عن طريق تغيير دلالات واجهته.
- تخرج هذا الطريقة بسرعة عن التحكم، عندما تريد أن توسعها على الصفوف الأخرى المشتقة. ماذا سيحدث عندما تجد قط بدون ذيل؟ أو قط لا يمسك الفئران؟ أو قط لا يشرب حليب؟ في النهاية سوف تنتهي بصفوف مشتقة مثل ScratchlessTaillessMicelessMilklessCat.
- مع مرور الوقت، تؤدي هذه الطريقة إلى الحصول على شفرة من المربك الحفاظ عليها، لأن الواجهات والسلوك للصفوف السلف تُلمح قليلاً أو لا تلمح حول سلوك أحفادها.

المكان لإصلاح هذه المشكلة ليس في صف القاعدة، ولكن في الصف الأصلي Cat. وتوليد صف حول المخالب Claws، واحتواءه داخل الصف Cats. كانت المشكلة الأساسية هي بافتراض أن جميع القطط لها مخالب، وبالتالي أصلح هذه المشكلة في مكان المصدر، بدلاً من فقط تضميدها في مكان الوجهة.

تجنب أشجار الوراثة العميقة تزوّد البرمجة غرضية التوجه عدد كبير من التقنيات لإدارة التعقيد. ولكن لكل أداة فعالة خطورتها، وبعض تقنيات البرمجة غرضية التوجه لديها ميل لزيادة التعقيد بدلاً من الحد منه.

اقترح آرثر ريل (Arthur Riel) في كتابه الممتاز "إرشادات التصميم غرضي التوجه-1996" (Object-Oriented Design Heuristics (1996))، الحد من التسلسل الهرمي للوراثة إلى حد أقصى بـ 6 طبقات. اعتمد ريل في توصياته على "العدد السحري 7±2"، ولكن أعتقد أن هذا متفائل بشكل كبير. استناداً إلى خبرتي، لدى معظم الناس مشكلة فهم أكثر من اثنين أو ثلاثة مستويات من الوراثة في أدمغتهم في نفس الوقت. من الممكن أن يكون "الرقم السحري 7±2" مُناسب أكثر كعدد الصفوف الفرعيّة للصف القاعدة، بدلاً من عدد المستويات في شجرة الوراثة.

لقد وُجد أن أشجار الوراثة العميقة مُرتبطة ارتباطا وثيقا بزيادة معدل الأخطاء (باسيلي، برياند، وميلو 1996) (Basili, Briand, and Melo 1996). يعرف السبب أي واحد حاول أن يصحح التعقيد في هرمية الوراثة. تزيد شجرات الوراثة العميقة التعقيد، وهذا هو بالضبط عكس ما ينبغي على الوراثة إنجازه (تخفيف التعقيد). ابقي المهمة التقنية الأساسية بالبال. وتأكد من أنك تستخدم الوراثة لتجنب تكرار الشفرة وتقليل التعقيد.

فضّل تعدد الأشكال لفحص النوع الشامل. إن تكرار عبارات case يقترح أحياناً استخدام الوراثة كخيار أفضل، على الرغم من أن هذا ليس صحيحاً دائماً. سنعرض هنا مثال كلاسيكي عن شفرة يتطلب أكثر من منهج البرمجة غرضية التوجه:

مثال بلغة البرمجة سي++ عن عبارة case التي يجب استبدالها على الأرجح بتعدد الأشكال

```
switch ( shape.type ) {
 case Shape_Circle:
 shape.DrawCircle();
 break;
 case Shape_Square:
 shape.DrawSquare();
 break;
...
}
```

في هذا المثال، إن الاستدعاءات للإجرائيات shape.DrawCircle() ،shape.DrawSquare(), يجب أن يتم استبدالها بإجرائية واحدة لها الاسم shape.Draw()، والتي يمكن استدعاءها بغض النظر عن الشكل دائرة(circle) أو مربع (square). من ناحية أخرى، تُستخدم أحياناً عبارات case لفصل أنواع مختلفة تماماً من الكائنات أو السلوك. هنا لدينا مثال حول عبارة case، التي تُعد مناسبة لبرنامج غرضي التوجه:

مثال بلغة البرمجة سي++ عن عبارة case، التي لا يجب استبدالها على الأرجح بتعدد الأشكال

```
switch ( ui.Command() ) {
 case Command_OpenFile:
 OpenFile();
 break;
 case Command_Print:
 Print();
 break;
 case Command_Save:
 Save();
 break;
 case Command_Exit:
 ShutDown();
 break;
...
}
```

في هذه الحالة، من الممكن تشكيل صف قاعدة مع صفوف مشتقة و إجرائية متعددة الأشكال DoCommand() لكل أمر(كما هو الأمر بالنسبة لنمط الأمر). ولكن في الحالات البسيطة كما هذه الحالة، معنى الإجرائية DoCommand() سيكون مخفف بشكل كبير كما لو كان بلا معنى، و عبارة الحالة case هي الحل أكثر قابلية للفهم.

أجعل جميع البيانات خاصة وليس محمية كما يقول جوشوا بلوخ (Joshua Bloch)، "الميراث يكسر التغليف". عندما ترث من كائن ما، فإنك تحصل على جميع امتيازات الوصول للإجرائيات المحمية والبيانات لهذا الكائن. إذا احتاج الصف المشتق حقاً الوصول لخواص الصف القاعدة، فاستخدم توابع ذات مستوى الوصول المحمي.

الوراثة المتعددة[1]:

الوراثة هي أداة فعالة. إنها مثل استخدام المنشار الآلي لقطع الشجر بدلاً من استخدام المنشار اليدوي. إنه من الممكن أن يكون مُفيد جدا عند استخدامه بحذر، ولكنه خطر في يد شخص لا يأخذ الاحتياطات المناسبة.

إذا كانت الوراثة هي منشار ألي، فإن الوراثة المتعددة هي منشار ألي من الخمسينات، بدون حماية على الشفرة، وبدون إغلاق أتوماتيكي، ومع محرك صعب. هنالك بعض الأوقات التي من الممكن أن تكون فيها هذا الأداة مفيدة، ولكن في معظم الأوقات من الأفضل تركها في الكراج في المكان الوحيد التي لن تستطع فيها أن تتسبب بضرر.

على الرغم من أنه ينصح بعض الخبراء بالاستخدام الواسع للوراثة المتعددة (ماير 1997) (Meyer 1997)، إلا أنه استناداً إلى خبرتي، الوراثة المتعددة مفيدة بالدرجة الأولى في تعريفات "mixins"، والصفوف البسيطة المُستخدمة لإضافة مجموعة من الخواص إلى كائن ما. تُدعى هذه الصفوف بـ " Mixins " لأنها تسمح لجميع الخواص بأن تكون مختلطة "mixed in" للصفوف المشتقة. من الممكن أن تكون الصفوف Displayable, Persistant, Serializable, Sortable هي صفوف Mixins. إن صفوف Mixins، تقريباً دائما هي صفوف مجردة وليس المقصود أن تكون مثبتة بشكل مستقل عن الكائنات الأخرى.

تتطلب Mixins استخدام الوراثة المتعددة، ولكنها لا تخضع لمشكلة الوراثة الكلاسيكية المرتبطة بالوراثة المتعددة، وذلك طالما أن صفوف Mixins هي بالحقيقة مستقلة عن بعضها البعض. كما أنها تجعل من التصميم أكثر قابلية للفهم، عن طريق تجميع الخواص مع بعض. سيملك المبرمج وقت أسهل لفهم أن كائن يستخدم الـ mixins، Displayable، Persistent)، أكثر من فهم أن الكائن يستخدم 11 إجرائية.

تقوم لغتي البرمجة جافا وفيجوال بيسك بتعريف قيم mixins عن طريق السماح بالوراثة المتعددة للواجهات ولكن فقط وراثة مفردة للصف. تدعم لغة البرمجة سي++ الوراثة المتعددة لكلا الواجهة والتنفيذ. يجب على

[1] حقيقة واحدة لا جدال فيها حول الوراثة المتعددة في سي ++ هي أنها تفتح صندوق باندورا من التعقيدات التي ببساطة لا وجود لها في الوراثة المفردة.- سكوت مايرز (Scott Meyers)

المبرمجين استخدام الوراثة المتعددة فقط بعد الأخذ بعين الاعتبار الحلول البديلة، وتقييم الأثر الذي سيحدث على تعقيد النظام والشمولية.

لماذا يوجد العديد من القواعد للوراثة؟



يُقدم هذا القسم العديد من القواعد لتجنب الوقوع في المشاكل عند العمل مع الوراثة. الرسالة الأساسية من كل هذه القواعد هي أن الوراثة تميل إلى العمل ضد الضرورة التقنية الأولية لديك كمبرمج، والتي هي إدارة التعقيد. من أجل السيطرة على التعقيد، يجب الحفاظ على التحيز الشديد ضد الوراثة. هنا سنعرض ملخص عن متى يجب استخدام الوراثة ومتى لا يجب استخدامها:

- إذا كانت عدة صفوف تشترك ببيانات مشتركة وليس بسلوك مشترك، عندها شكل كائن مشترك، الذي من الممكن أن يحوي هذه الصفوف[1].
- إذا كانت عدة صفوف تشترك بسلوك مشترك وليس ببيانات مشتركة، عندها اشتق هذه الصفوف من صف القاعدة، الذي يعرّف الإجرائيات المشتركة.
- إذا كانت عدة صفوف تشترك ببيانات وبسلوك مشترك، عندها أورث من صف القاعدة، الذي يعرّف البيانات والإجرائيات المشتركة.
- أورث عندما تريد من الصف القاعدة أن يتحكم بواجهتك؛ احتوي عندما تريد أن تتحكم بواجهتك.

توابع العنصر والبيانات

هنا يوجد بضع التوجيهات حول تنفيذ توابع العنصر وبيانات العنصر بشكل فعّال[2].

حافظ على عدد الإجرائيات في الصف صغير قدر الإمكان وُجد من دراسة البرامج المكتوبة بلغة البرمجة سي++، أن زيادة عدد الإجرائيات في الصف مرتبط بزيادة معدل الأخطاء (باسيلي، برياند، وميلو 1996) (Basili, Briand, and Melo 1996). على كل حال، ووُجد أيضاً عوامل منافسة أخرى، قد تكون أكثر أهمية، مثل عمق شجرة الوراثة، والعدد الكبير للإجرائيات المستدعية داخل الصف، والترابط القوي بين الصفوف. قيّم المقايضة بين إنقاص عدد الإجرائيات وهذه العوامل الأخرى.

[1] **إشارة مرجعية**: للمزيد حول التعقيد، انظر "الضرورة التقنية الأساسية للبرمجيات: إدارة التعقيد" في القسم 5-2.

[2] **إشارة مرجعية**: للمزيد حول المناقشات حول الإجرائية بشكل عام، انظر الفصل 7 "الإجرائيات عالية الجودة"

ارفض توابع العنصر المولدة ضمنياً، والمعاملات التي لا تحتاجها في بعض الأوقات ستجد أنك ترغب برفض توابع معينه- من الممكن أنك ترغب برفض الاسناد، أو من الممكن ألا ترغب بالسماح ببناء كائن ما. ومن الممكن أن تفكّر بهذه الطريقة، بما أن المترجم البرمجي يولد المعاملات بشكل أتوماتيكي، فلا يمكنك بالسماح للوصول. ولكن في هذه الحالات، يمكنك أن ترفض تلك الاستخدامات، عن طريق التصريح عن باني أو معامل اسناد، أو توابع أخرى أو المعامل `private`، الذي سيمنع الزبون من الوصول إليه. (إن جعل الباني خاص `private`، هي عبارة عن تقنية قياسية لتعريف الصف المفرد، والذي سيتم مناقشته في هذا الفصل).

قلّل من عدد الإجرائيات المختلفة المُستدعاه من قبل صف وجدت أحد الدراسات أن عدد الأخطاء في صف مرتبطة بشكل كبير مع العدد الكلي للإجرائيات التي تم استدعائها داخل الصف (باسيلي، برياند، وميلو 1996) (Basili, Briand, and Melo 1996). ولقد وجدت نفس الدراسة أنه كلما كان عدد الصفوف التي يستخدمها صف كبير، كلما مال معدل خطأ هذا الصف إلى الأعلى. تُسمى هذه المفاهيم أحيانا "fan out".

قلّل من استدعاءات الإجرائية لصفوف أخرى إن الاتصالات المباشرة خطرة بما فيه الكفاية[1]. أما الاتصالات غير المباشرة– مثل

```
account.ContactPerson().DaytimeContactInfo().
PhoneNumber()
```

– تميل إلى أن تكون أكثر خطورة. لقد صاغ الباحثين قاعدة تُدعى "قانون ديميتر" (ليبرهير وهولندا 1989) ("Law of Demeter" (Lieberherr and Holland 1989))، التي تنص أساساً على أن الكائن "أ" يمكنه استدعاء أي من إجراءاته التكرارية الخاصة. إذا قام الكائن "أ" بإنشاء الكائن "ب"، فإنه يمكنه أن يستدعي أي من الإجرائيات للكائن "ب". ولكن عليه تجنب استدعاء الإجرائيات على الكائنات المزودة من الكائن "ب". في مثال الحساب فوق، إن الاستدعاء

```
account.ContactPerson()
```

صحيح، أما الاستدعاء

```
account.ContactPerson().DaytimeContactInfo()
```

غير صحيح

هذا عبارة عن شرح بسيط، لمزيد من التفاصيل، انظر المصادر الإضافية في نهاية هذا الفصل.

بشكل عام، قلل من المدى الذي فيه يتعاون الصف مع صفوف أخرى. حاول أن تقلل من التالي:

[1] **اقرأ أيضاً** يمكن العثور على حسابات جيدة من قانون ديميتر (Law of Demeter) في المبرمج العملي (هانت وتوماس 2000 (Pragmatic Programmer (Hunt and Thomas 2000))، وتطبيق لغة النمذجة الموحدة والنماذج (لارمان 2001 (Applying UML and Patterns (Larman 2001))، وأساسيات التصميم غرضي التوجه في لغة النمذجة الموحدة (بيج-جونز 2000 (Fundamentals of Object-Oriented Design in UML (Page-Jones 2000)).

- عدد أنوع الكائنات الممثلة
- عدد استدعاءات المباشرة المختلفة للإجرائية في الكائنات الممثلة
- عدد استدعاءات الإجرائيات المعادة من الكائنات الممثلة الأخرى

البواني Constructors

يُمثل التالي مبادئ توجيهية مُخصصة للتطبيق على البواني (constructors). إن هذه المبادئ التوجيهية مشابهة وقابلة للتطبيق لمجوعة من اللغات (سي++، وجافا، وفي كل مكان). ولكن تختلف الهوادم (Destructors) بشكل أكبر، لذلك عليك أن تتفحص المواد الموجودة في فصل "المصادر الإضافية"، قسم المعلومات حول الهوادم.

هيّء كل عناصر البيانات في كل البواني - إذا كان هذا ممكن. إن تهيئة كل عناصر البيانات في كل البواني هي ممارسة برمجة دفاعية غير مكلفة.

افرض خاصية الوليد المفرد[1] (singleton)، عن طريق استخدام الباني الخاص إذا كنت ترغب بتعريف صف يسمح بتشكيل فقط كائن وحيد، فيمكنك أن تفرض هذا عن طريق إخفاء جميع البواني للصف، ومن ثم تقوم بتزويد الإجرائية الساكنة GetInstance، للوصول إلى الكائن الوحيد من الصف.

مثال بلغة البرمجة جافا عن فرض خاصية الوليد المفرد (singleton) عن طريق استخدام الباني الخاص.

```
public class MaxId {
 // البواني و المهدمات
 private MaxId() {
...
 }
...
 // الإجرائيات العامة

public static MaxId GetInstance() {
 return m_instance;
 }
...
 // العناصر الخاصة
```

هنا الباني الخاص.

هنا الإجرائية العامة التي تؤمن الوصول إلى نسخة واحدة من الصف

[1] **اقرأ أيضا**: الشفرة التي تفعل هذا في لغة البرمجة سي++ ستكون مُشابهة. لمزيد من التفاصيل، انظر "سي++ أكثر فعالية " البند 26 (مايرز 1998).

```
 private static final MaxId m_instance = new MaxId();
...
}
```

هنا النسخة الوحيدة (instance)

يتم استدعاء الباني الخاص فقط عندما يتم تشكيل الكائن الساكن m_instance. في هذه الطريقة إذا كنت تريد أن تشير إلى الوليد المفرد (الكائن المفرد المُشكل) MaxId، يمكنك ببساطة أن تشير إلى الإجرائية MaxId.GetInstance().

فضّل النسخ العميقة عن النسخ السطحية، حتى يثبت خلاف ذلك إن أحد القرارات الصعبة التي ستتخذها حول الكائنات المعقدة، هو قرار تنفيذ نسخ عميقة أو نسخ سطحية لهذا الكائن. تُعتبر النسخة العميقة للكائن، كنسخة لبيانات الكائن؛ أما النسخة السطحية فقط تُشير إلى نسخة مرجعية مفردة، على الرغم من اختلاف المعنى التخصصي للكلمتين "عميق" و "سطحي".

إن الدافع لتشكيل نسخة سطحية هو بشكل نموذجي لتحسين الأداء. على الرغم من أن إنشاء نسخ متعددة من الكائنات الكبيرة قد يكون عمل ثقيل، فإنه نادراً ما يسبب أي تأثير يمكن قياسه على الأداء. من الممكن أن يسبب عدد صغير من الكائنات مشاكل في الأداء، ولكن المبرمجين ضعيفين في تخمين أي شفرة في الحقيقة تُسبب المشاكل. (لمزيد من التفاصيل، انظر الفصل 25 "استراتيجيات ضبط الشفرات".)

بما أن إضافة التعقيد على مكاسب الأداء المشكوك فيها تُعتبر مقايضة ضعيفة، فإن النهج الصحيح بين النسخ العميقة والنسخ السطحية، هي بتفضيل النسخ العميقة حتى يتم إثبات أن العكس صحيح.

من السهل كتابة الشفرة للنسخ العميقة والمحافظة عليها، بالمقارنة بالنسخ السطحية. بالإضافة إلى الشفرة، التي تحوي أي نوع من الكائنات، تُضيف النسخ السطحية شفرة إلى المراجع، وتضمن نسخ كائن أمنة، ومقارنات آمنة، وحذوفات أمنة، وإلى أخره. وستكون هذه الشفرة معرضة للخطأ، وعليك أن تتجنب تشكيلها، إلا إذا وجد سبب قهري لتشكيله.

إذا وجدت أنك بحاجة فعلاً إلى استخدام نهج النسخة السطحية، يحوي كتاب سكوت مايرز " سي++ أكثر فعالية"، البند 29 (1996) (Scott Meyers's More Effective سي++1996) , Item 29)) على مناقشة ممتازة لقضايا لغة البرمجة سي++. يصف مارتن فاولر في كتابه "إعادة بناء التعليمات البرمجية (1999)" (Martin Fowler's Refactoring (1999)) الخطوات الخاصة للتحويل من النسخ السطحية إلى النسخ العميقة، ومن العميقة إلى السطحية. (فاولر يطلق عليها الكائنات المرجعية وكائنات القيمة).

4.6 أسباب تشكيل صف[1]

إذا صدقت كل ما قرأته، من الممكن أن تفكر بأن السبب الوحيد لتشكيل صف هو نمذجة كيانات العالم الحقيقي. ولكن في الممارسة العملية، سوف تجد أن الصفوف تُشكل لأسباب أكثر من هذا السبب. سنعرض هنا قائمة بالأسباب الجيدة لتشكيل صف.

نمذجة كيانات العالم الحقيقي[2] إن نمذجة كيانات العالم الحقيقي، من الممكن أن يكون ليس السبب الوحيد لتشكيل الصفوف ولكنه لا يزال سبب جيد. ضع بيانات كيان العالم الحقيقي في صف، ومن ثم شكل إجرائيات الخدمة التي تنمذج سلوك هذه الكيان. انظر إلى المناقشات حول أنواع البيانات المجردة في القسم 6-1 من أجل الأمثلة الموجودة فيه.

نموذج الكيانات المجردة. إن نمذجة الكيانات المجردة هو سبب جيد لتشكيل صف، الكيان المجرد- هو ليس بكيان ملموس، ليس بكيان العالم الحقيقي، ولكنه يزود تجريدية حول الكيانات الملموسة الأخرى. مثال جيد هو كيان الشكل الكلاسيكي Shape. حيث يوجد بشكل حقيقي الدائرة `Circle` والمربع `Square`، ولكن الشكل Shape هو عبارة عن تجريدية للأشكال المتخصصة الأخرى.

في مشاريع البرمجة، ليست التجريدية جاهزة كما هو في مثال الشكل Shape، بل يجب العمل بجدية أكبر للوصول إلى تجريدية واضحة نظيفة. إن عملية الحصول على مفاهيم التجريدية من كيانات العالم الحقيقي ليس بعملية قطعية، أي مصممين مختلفين سيحددون التجريد بعموميات مختلفة. إذا لم نكن نعرف حول الأشكال الهندسية مثل الدائرة والمربع والمثلث، على سبيل المثال، من الممكن أن نأتي بأشكال غير اعتيادية مثل شكل القرع، وشكل اللفت الأصفر، وشكل السيارة `Pontiac Aztek`. إن الحصول على الكائنات المجردة المناسبة، يُعد من التحديات الرئيسية في البرمجة غرضية التوجه.

نقطة مفتاحية

إنقاص التعقيد. إن السبب الوحيد الأكثر أهمية لتشكيل الصف هو بإنقاص تعقيد البرنامج. شكّل صف لإخفاء المعلومات، وبهذه الطريقة لن تكون بحاجة إلى التفكير بهذه المعلومات. بالتأكيد، ستحتاج إلى التفكير في هذه المعلومات عند تشكيل الصف. ولكن بعد كتابة الصف، عليك أن تكون قادر على نسيان التفاصيل، واستخدام الصف بدون معرفة أعماله الداخلية. إن الأسباب الأخرى لتشكيل الصفوف- تقليل حجم الشفرة، تحسين قابلية الصيانة، تحسين التصويب- أيضاً تُعد أسباب جيدة، ولكن بدون القوة التجريدية للصفوف، إن البرامج المعقدة سيكون من المستحيل إدارتها فكرياً.

[1] **إشارة مرجعية** تداخل أسباب تشكيل الصفوف والإجرائيات- القسم 7-1

[2] **إشارة مرجعية** لمزيد من تعريف كيانات العالم الحقيقي، انظر "البحث عن كيانات العالم الحقيقي" في القسم 5-3

عزل التعقيد. إن التعقيد في كل صيغه- الخوارزميات المعقدة، ومجموعات البيانات الكبيرة، وبرتوكولات الاتصالات المعقدة، وإلى أخره- مُعرض للأخطاء. إذا حدث خطأ، سيكون من السهل إيجاده إذا كان هذا الخطأ يوجد داخل صف وليس منتشر خلال كل الشفرة. لن تؤثر التغيرات الناشئة من إصلاح الخطأ على بقية الشفرة، لأنه سيتصلح فقط صف واحد- ولن يتم لمس الشفرة الباقية. إذا وجدت على سبيل المثال خوارزمية أفضل وأبسط وأكثر موثوقية، فسيكون من السهل استبدال الخوارزمية القديمة، إذا كانت الخوارزمية القديمة معزولة داخل صف. وخلال عملية التطوير، سيكون من الأسهل تجريب عدة تصاميم، والمحافظة على التصميم الذي يعمل بشكل أفضل.

إخفاء تفاصيل التنفيذ. تُعد الرغبة في إخفاء تفاصيل التنفيذ سبب رائع لتشكيل الصف، سواء كانت التفاصيل معقدة كالوصول المعقد إلى قاعدة البيانات، أو عادية غير معقدة، كعنصر بيانات متخصص مُخزن كعدد أو كسلسلة محرفية.

الحد من آثار التغييرات يحد عزل المناطق التي من المرجح أن تتغير، من أثار التغييرات إلى مجال واحد من صف مفرد أو عدة صفوف. ويصبح تصميم تلك المناطق التي من المرجح أن تتغير كلياً، سهل التغيير. تتضمن المناطق التي من المرجح أن تتغير على أجهزة ملحقة، دخل/خرج، أنواع بيانات معقدة وعلى قواعد الأعمال. تصف الأقسام الفرعية "إخفاء الأسرار (إخفاء المعلومات)" في القسم 5-3 المصادر الشائعة المختلفة للتغيير.

إخفاء البيانات الشاملة[1]. إذا كنت في حاجة إلى استخدام البيانات الشاملة، يمكنك إخفاء تفاصيل تنفيذها خلف واجهة صف. يوفر العمل مع البيانات الشاملة من خلال إجرائيات الوصول الكثير من الفوائد، بالمقارنة مع العمل مع البيانات الشاملة بشكل مباشر. يمكنك تغيير هيكلية البيانات، بدون تغيير برنامجك. يمكنك مراقبة الوصولات للبيانات. يُشجعك أيضاً مبدأ إجرائيات الوصول على التفكير فيما إذا كانت البيانات شاملة فعلاً، حيث يتضح في كثير من الأحيان أن "البيانات الشاملة " هي فعلاً فقط بيانات الكائن.

تبسيط تمرير الوسيط. إذا كنت تمرر وسيط بين مجموعة من الإجرائيات، التي من الممكن أن تحتاج أن تضع هذه الإجرائيات داخل صف يشارك الوسيط كبيانات كائن. إن تبسيط تمرير الوسيط ليس بهدف، في حد ذاته، ولكن يقترح تمرير الكثير من البيانات إلى تنظيم صف مختلف، من الممكن أن يعمل بشكل أفضل.

[1] **إشارة مرجعية** للمزيد من المناقشات حول المشاكل المرتبطة باستخدام البيانات الشاملة، انظر القسم 13-3، "البيانات الشاملة "

جعل نقاط مركزية للسيطرة[1]. إنها لفكرة جيدة التحكم في كل مهمة في مكان واحد. يفترض التحكم العديد من الصيغ. معرفة عدد عناصر الإدخالات في جدول ما، هي إحدى الصيغ. التحكم بالأجهزة- الملفات، روابط قواعد البيانات، الطوابع، وإلى أخره- هو صيغة أخرى للتحكم. إن استخدام صف للقراءة والكتابة في قاعدة بيانات هو صيغة من صيغ التحكم المركزي. إذا كان هناك في حاجة إلى تحويل قاعدة البيانات إلى ملف ثابت أو إلى بيانات في الذاكرة، فإن هذا التغييرات سوف تؤثر فقط على صف وحيد. إن فكرة التحكم المركزي مشابهه لإخفاء المعلومات، ولكنها تملك استطاعة استكشافية فريدة من نوعها، لذلك إنها تستحق أن يتم إضافتها إلى صندوق أدواتك للبرمجة

تسهيل عملية إعادة استخدام الشفرة يمكن إعادة استخدام الشفرة الموضوعة في صفوف جيدة في برامج أخرى بسهولة أكبر من تضمين نفس الشفرة في صف واحد كبير. حتى لو كانت الشفرة مُستدعاه من قبل مكان واحد في البرنامج، فإنها مفهومة كجزء من صف أكبر، فإنه من الأفضل في حالة استخدام هذا القسم من الشفرة في برنامج أخر، وضع هذا القسم من الشفرة في صف خاص به.

بيانات مثبتة

درس مختبر هندسة البرمجيات التابع لناسا عشرة مشاريع تسعى لإعادة استخدامها (مغاري، واليغورا، وماكديرموت 1989) (McGarry, Waligora, and McDermott 1989). باستخدام كلا المنهجين غرضي التوجه ووظيفي التوجه، لم يستطيعوا في البرامج الأولية أخذ الكثير من شفرة المشاريع السابقة، وذلك لأن المشاريع السابقة لم تكن قد اسست قاعدة شفرة كافية. بعد ذلك، كان هناك إمكانية أخذ 35 بالمئة من شفرة المشاريع السابقة التي استخدمت التصميم الوظيفي. وكان هناك إمكانية لأخذ 70 بالمئة أو أكثر من شفرة المشاريع السابقة التي استخدمت المنهج غرضي التوجه. إذا كان هناك إمكانية لتجنب إعادة كتابة 70 بالمئة من برنامجك عن طريق التخطيط المسبق، فقم بذلك!

الجدير بالذكر، أن جوهر نهج ناسا في إنشاء الصفوف القابلة لإعادة الاستخدام لا ينطوي على "التصميم لإعادة الاستخدام".[2] حددت ناسا ترشيحات إعادة الاستخدام عند نهاية مشاريعها. ومن ثم قاموا بأداء العمل المطلوب لجعل الصفوف قابلة لإعادة الاستخدام كمشروع خاص في نهاية المشروع الأساسي، كخطوة جديدة لمشروع جديد. تساعد هذه الطريقة بمنع "الطلي بالذهب" "gold-plating" – الذي هو عبارة عن توليد وظيفية غير مطلوبة وتضيف تعقيد غير ضروري.

---

[1] **إشارة مرجعية** لمزيد من التفاصيل حول إخفاء المعلومات، انظر إخفاء الأسرار (إخفاء المعلومات) في القسم 5-3

[2] **إشارة مرجعية** للمزيد حول تنفيذ الكمية الأصغرية من الوظيفية المطلوبة، انظر " يحتوي البرنامج على شفرة يبدو أنه قد تكون هناك حاجة إليه في يوم من الأيام " في القسم 24-2

خطة لعائلة من البرامج. إذا كنت تتوقع أن يتم تعديل برنامج، فإنها لفكرة جيدة عزل الأجزاء التي من الممكن أن تتغير في صفوف خاصة بها. ومن ثم تستطيع أن تعدل الصفوف بدون التأثير على باقي البرنامج، أو تستطيع وضعه بشكل كامل في صفوف جديدة. يعتبر التفكير ليس فقط بكيف سيبدو البرنامج، بل التفكير كيف ستبدو عائلة كاملة من البرامج، عبارة عن استدلال فعال لتوقع كامل الأصناف من التغيرات (برناس 1976) (Parnas 1976).

منذ عدة سنوات، قمت بإدارة فريق لكتابة سلسلة من البرامج المستخدمة من قبل زبائنا لشراء بوليصات التأمين. كان علينا أن نضم إلى كل برنامج معدلات التأمين الخاصة للعميل، وصيغة التقرير، وإلى أخره. ولكن العديد من أجزاء البرنامج كانت متشابهة: الصفوف التي تدخل معلومات حول الزبائن المحتملين، والصفوف التي تخزن المعلومات في قاعدة بيانات الزبون، والصفوف التي تبحث عن المعدلات، والصفوف التي تحسب المعدلات الكلية لمجموعة، وإلى أخره. قام الفريق بتجزئة البرنامج، وبالتالي كل جزء اختلف من زبون إلى أخر في صفه الخاص. أخذت البرمجة الأولية تقريباً ثلاثة أشهر أو أكثر، ولكن عندما حصلنا على زبون جديد، فقط قمنا بكتابة بضع صفوف جديدة للزبون الجديد، وحافظنا على بقية الشفرة كما هي. بضعة أيام عمل، وها هو البرنامج جاهز.

حزّم العمليات المتصلة مع بعضها. في الحالات التي لا تستطيع فيها إخفاء المعلومات، مشاركة البيانات، أو التخطيط للمرونة، لا تزال تستطيع رزم مجموعة من العمليات داخل مجموعات معقولة، مثل توابع علم المثلثات، والتوابع الإحصائية، وإجرائيات التلاعب بالسلاسل المحرفية، وإجرائيات التلاعب بالبتات، وإجرائيات الرسومات، وإلى أخره. إن الصفوف هي إحدى وسائل تجميع العمليات المرتبطة مع بعضها. تستطيع استخدام التحزيم، أو اسماء النطاقات، أو الملفات الرأسية، وذلك بالاعتماد على أي لغة تعمل عليها.

إنجاز إعادة هيكلة محددة. تنتج العديد من صفات إعادة بناء التعليمات البرمجية الموصوفة في الفصل 24 "إعادة التصنيع"، في صفوف جديدة- متضمنة تحويل صف واحد إلى صفين، إخفاء المفوض، حذف الشخص الوسيط، وإدخال صف ممدد. من الممكن لهذه الصفوف الجديدة أن تعدل، عند وجود رغبة لإنجاز الأهداف المشروحة خلال هذا الفصل.

الصفوف التي يجب تجنبها

على الرغم من أن الصفوف بشكل عام جيدة، من الممكن أن تنخدع في بعضها. هنا سنتحدث عن الصفوف التي يجب تجنبها.

تجنب تشكيل صفوف عملاقة. تجنب تشكيل صفوف عملاقة، تعرف كل شيء وقادرة على كل شيء. إذا كان الصف يصرف وقته في ا سترجاع البيانات من صفوف أخرى با ستخدام الإجرائيات () Get و()Set (أي الغوص في أعمالهم ويقول لهم ماذا يفعلون)، عليك أن تسـأل فيما إذا كان أفضـل تنظيم هذه الوظيفية في صـفوف أخرى بدلاً من استخدام صف عملاق (ريبل 1996) (Riel 1996).

أزل الصفوف عديمة الصلة[1] إذا كان الصف يحوي فقط على بيانات بدون سلوك، اسأل نفسك فيما إذا كان هذا حقاً صف، وفكّر في امكانية تخفيض مستواه، بحيث يصبح عنصر البيانات فيه فقط صفات لصف أخر أو عدّة صفوف أخرى.

تجنب الصفوف المسماة بعد الأفعال إن الصف الذي يملك فقط سلوك بدون بيانات هو في الحقيقة ليس بصف. عندها فكّر بتحويل هذا الصف مثل DatabaseInitialization أو Builder إلى إجرائية لبعض الصفوف الأخرى.

ملخص لأسباب إنشاء الصفوف

هنا نستعرض قائمة تلخص الأسباب المتاحة لتشكيل صف:

- نمذجة كيانات العالم الحقيقي
- نمذجة الكيانات المجردة
- إنقاص التعقيد
- عزل التعقيد
- إخفاء تفاصيل التنفيذ
- الحد من آثار التغييرات
- إخفاء البيانات الشاملة
- تبسيط تمرير الوسطاء
- جعل نقاط مركزية للسيطرة
- تسهيل عملية إعادة استخدام الشفرة
- خطة لعائلة من البرامج

[1] **إشارة مرجعية**: يُدعى عادةً هذا النوع من الصفوف بالهيكل. لمزيد من المعلومات حول الهياكل، انظر القسم 13-3 "الهياكل".

- حزم العمليات المتصلة مع بعضها
- إنجاز إعادة هيكلة محددة

5.6 مشاكل لغة برمجة محددة

تتغير طرق التعامل مع الصفوف في لغات برمجة مختلفة بطرق مثيرة للاهتمام. اعتبر أنك تُعيد تعيين إجرائية لإنجاز تعدد الأشكال في صف مشتق. في لغة البرمجة جافا، افتراضياً كل الإجرائيات قابلة لإعادة التعيين، ولمنع الصف المشتق من إعادة تعيين إجرائية ما، يجب التصريح عن هذه الإجرائية كإجرائية نهائية باستخدام الكلمة المفتاحية `final`. أما في لغة البرمجة سي++، افتراضياً كل الإجرائيات غير قابلة لإعادة التعيين. ولجعل الإجرائية قابلة لإعادة التعيين يجب التصريح عنها بشكل افتراضي في الصف القاعدة. في لغة البرمجة فيجوال بيسك، يجب أن يتم التصريح عن إجرائية بأنها قابلة لإعادة التعيين في الصف القاعدة، ويجب على الصف المشتق استخدام الكلمة المفتاحية لإعادة التعيين.

هنا سنعرض المجالات المرتبطة بالصف والتي تتغير بشكل كبير بالاعتماد على أي لغة برمجة مستخدمة:

- سلوك البواني والهوادم المعاد تعيينها في شجرة الوراثة
- سلوك البواني والهوادم في ظل ظروف استثنائية
- أهمية البواني الافتراضية (بواني بدون معاملات)
- وقت استدعاء الهادم أو الناهي terminator.
- الحكمة من إعادة تعيين العمليات المبنية في اللغة نفسها، بما في ذلك عمليات الإسناد والمساواة.
- ما هو حجم الذاكرة الذي يتعامل معه عند تشـكيل الكائنات أو عند هدمها، أو عند التصـريح عنها ومن ثم الخروج من النطاق.

إن المناقشات التفصيلية لهذه المشاكل هي خارج نطاق هذا الكتاب، ولكن يشير قسم "المراجع الإضافية" إلى مصادر جيدة مختصة بكل لغة.

6.6 الصفوف الخلفية: الرزم[1]

[1] **إشارة مرجعية** لمزيد من المعلومات حول التمييز بين الصفوف والحزم، انظر "مستويات التصميم" في القسم 5.

إن الصفوف حالياً هي أفضل طريقة للمبرمجين لتحقيق النمطية. ولكن النمطية موضوع كبير، ويمتد إلى ما خلف الصفوف. على مر السنوات الماضية، لقد تطورت البرمجيات في جزء كبير، عن طريق زيادة تقسيم التجمعات التي علينا العمل معها. أن أول تجمع حصلنا عليه هو العبارة البرمجية، والتي في ذلك الوقت بدى وكأنه خطوة كبيرة في مجال تعليمات الألة. ومن ثم ظهرت الوظائف الفرعية، ومن ثم الصفوف.

من الواضح، أننا لا نستطيع دعم أهداف التجريدية والتغليف بشكل جيد، إذا لم نملك أدوات جيدة لمجموعات التجميع للكائنات. أيدت لغة البرمجة Ada مفهوم الرزم قبل أكثر من عقد من الزمان، واليوم تؤيد لغة البرمجة جافا مفهوم الرزم. إذا كنت تبرمج في لغة لا تدعم مفهوم الرزم بشكل مباشر، فإنه من الممكن أن تشكل نسختك الضعيفة من رزمة وتنفيذها من خلال معايير البرمجة التي تشمل ما يلي:

- تسمية الاتفاقيات، التي تميز بين الصفوف العامة وتلك التي تُستخدم للاستخدام الخاص للرزمة.
- تســمية الاتفاقيات، اتفاقيات تنظيم الشــفرة (هيكل البرنامج)، أو كلاهما من أجل تحديد لأي رزمة ينتمي كل صف.
- القواعد التي تحدد أية رزم يُسـمح اسـتخدامها من قبل رزم أخرى، بما في ذلك ما إذا كان الاسـتخدام يمكن أن يكون الوراثة، أو الاحتواء، أو كلاهما.

تُعد هذه الحلول أمثلة جيدة للتمييز بين البرمجة إلى لغة مقابل البرمجة في اللغة.. لمزيد من المعلومات حول هذا التمييز، انظر القسم 34-3، "برمج داخل لغتك وليس إليها".

لائحة اختبار1: جودة الصف

أنواع البيانات المجردة2

- هل فكرت بالصــفوف في برنامجك كأنواع بيانات مجرّدة وقيّمت واجهاتها من وجهة النظر هذه؟

التجريدية

- هل يملك الصف غرض مركزي؟
- هل الصف مُسمى بشكل جيد، وهل يصف اسمه الغرض المركزي منه؟
- هل تقدم واجهة الصف تجريدية متسقة؟
- هل توضح واجهة الصف الكيفية لاستخدام الصف؟

[1] cc2e.com/0672

[2] **إشارة مرجعية**: هذه عبارة عن قائمة التحقق للاعتبارات حول جودة الصف. للحصول على قائمة بالخطوات المستخدمة لبناء الصف، انظر قائمة التحقق "برمجة الشفرة الزائفة" في القسم 9، الصفحة 233.

- هل واجهة الصف مجردة بما فيه الكفاية بحيث ليس عليك أن تفكّر بكيف يتم تنفيذ خدماتها؟ هل تستطيع التعامل مع الصف كصندوق أسود؟
- هل خدمات الصف كاملة بما فيه الكفاية، بحيث ليس هناك ضرورة لتدخل الصفوف الأخرى ببياناتها الداخلية؟
- هل تم استبعاد المعلومات غير المتصلة من الصف؟
- هل فكّرت حول تقسيم الصف إلى الصفوف المكونة، وهل قمت بتقسيمها قدر الإمكان؟
- هل تحافظ على سلامة واجهة الصف عندما تقوم بتعديل الصف؟

التغليف

- هل يقلل الصف من قابلية الوصول لعناصره؟
- هل يتجنب الصف عرض عنصر البيانات؟
- هل يخفي الصف تفاصيل تنفيذه عن الصفوف الأخرى بقدر ما تسمح لغة البرمجة؟
- هل يتجنب الصف صنع افتراضات حول مستخدميه، بما في ذلك صفوفه المشتقة؟
- هل الصف مستقل عن الصفوف الأخرى؟ هل هو متصل معهم بمقدار ضئيل؟

الوراثة

- هل يتم استخدام الوراثة فقط لنمذجة العلاقات "هو"- أي هل تلتزم الصفوف المشتقة بمبدأ ليسكوف حول الاستبدال؟
- هل يصف توثيق الصف استراتيجية الوراثة؟
- هل تتجنب الصفوف المشتقة "إعادة التعيين" للإجرائيات غير القابلة لإعادة التعيين.
- هل الواجهات والبيانات والسلوكيات المشتركة عالية قدر الإمكان في شجرة الوراثة؟
- هل أشجار الوراثة سطحية إلى حد ما؟
- هل كل عناصر البيانات في الصف القاعدة خاصة بدلاً من محمية؟

مشاكل التنفيذ الأخرى

- هل يحتوي الصف حوالي سبع عناصر بيانات أو أقل؟
- هل يقلل الصف من استدعاءات الصف المباشرة وغير المباشرة للصفوف الأخرى؟
- هل يتعاون الصف مع الصفوف الأخرى فقط عند الضرورة القصوى؟
- هل تمت تهيئة كل عناصر البيانات في الباني؟

- هل تم تصميم الصف ليتم استخدامه كنسخ عميقة بدلاً من نسخ سطحية، إلا إذا كان هناك سبب مقنع لإنشاء نسخ سطحية؟

المشكلات الخاصة باللغة

- هل قمت بالتحقق من المشاكل المتعلقة بلغة محددة حول الصفوف في لغة البرمجة الخاصة بك؟

مصادر إضافية

الصفوف بشكل عام

ماير، برتراند. بناء البرمجيات غرضية التوجه، الإصدار الثاني. نيويورك،1 نيويورك: برنتيس هول بتر، 1997. Meyer, Bertrand. Object-Oriented Software Construction, 2d ed. New York, NY: Prentice Hall PTR, 1997.

يحتوي هذا الكتاب على مناقشة عميقة حول أنواع البيانات المجردة و يشرح الطريقة التي يصيغون فيها الأساس للصفوف. تناقش الفصول 14-16 الوراثة بعمق. يزود ماير إثباتات لصالح الوراثة المتعددة في الفصل 15.

ريل، آرثر ج. استدلالات التصميم غرضي التوجه. قراءة، ماساتشوستس: أديسون-ويسلي، 1996. Riel, Arthur J. Object-Oriented Design Heuristics. Reading, MA: Addison-Wesley, 1996.

يحتوي هذا الكتاب على قائمة بالاقتراحات لتطوير تصميم البرامج، معظمها على مستوى الصف. لقد تجنبت هذا الكتاب عدة سنوات، لأنه لشيء كبير أن تتكلم حول الناس في بيت زجاجي. على كل حال، يتألف الكتاب من 200 صفحة. تُعد كتابات ريل سهلة الفهم ومتعه. والمحتوى عملي ومركّز.

سي++

مايرز، سكوت سي++ الفعال: 50 طريقة محددة لتحسين برامجك وصميماتك، النسخة الثانية. قراءة، ماساتشوستس: أديسون-ويسلي، 21998.

Meyers, Scott. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, 2d ed. Reading, MA: Addison-Wesley, 1998.

[1] cc2e.com/0679
[2] cc2e.com/0686

مايرز، سكوت، 1996، سي++ الأكثر فعالية:35 طريقة جديدة لتحسين برامجك وتصميماتك. قراءة، ماساتشوستس: أديسون-ويسلي، 1996.

`Meyers, Scott, 1996, More EffectiveC++: 35 New Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley, 1996.`

يعد كلا كتابي مايرز مراجع أساسية لمبرمج سي++. إن الكتابان مسليين ويساعدان على إدراك لغة البرمجة سي++ بأدق تفاصيلها.

جافا

بلوش، جوشوا. الدليل الفعال للغة البرمجة جافا. بوسطن، ماساتشوستس: أديسون-ويسلي، 2001.[1]

`Bloch, Joshua. Effective Java Programming Language Guide. Boston, MA: Addison-Wesley, 2001.`

يوفر كتاب بلوش نصائح جيدة جداً حول لغة البرمجة جافا وكذلك يُعد كمقدمة عامة، ويقدم تدريبات عماية جيدة حول البرمجة غرضية التوجه.

فيجوال بيسك

إن الكتب التالية هي مراجع جيدة عن الصفوف في لغة البرمجة فيجوال بيسك[2]:

فوكسال، جيمس. المعايير العملية لـ `Microsoft Visual Basic`،NET، ريدموند، واشنطن: ميكروسوفت بريس، 2003.

`Foxall, James. Practical Standards for Microsoft Visual Basic.NET. Redmond, WA: Microsoft Press, 2003.`

كورنيل، غاري، وجوناثان موريسون. البرمجة `VB.NET`: دليل للمبرمجين ذوي الخبرة. بيركلي، كاليفورنيا: أبريس، 2002.

`Cornell, Gary, and Jonathan Morrison. Programming VB.NET: A Guide for Experienced Programmers. Berkeley, CA: Apress, 2002.`

بارويل، فريد، وآخرون. `VB.NET` المحترف، الإصدار الثاني. وروكس، 2002.

`Barwell, Fred, et al. Professional VB.NET, 2d ed. Wrox, 2002.`

نقاط مفتاحية

- يجب على واجهات الصف أن تؤمن تجريدية متناسقة. إن انتهاك هذا المبدأ يؤدي إلى ظهور العديد من المشاكل.

[1] cc2e.com/0693
[2] cc2e.com/0600

- يجب على واجهة صف أن تخفي شيء ما – واجهة نظام أو صنع قرار، أو تفاصيل التنفيذ.
- عادةً يُفضل الاحتواء على الوراثة، إلا إذا كنت تنمذج العلاقة "is a".
- إن الوراثة أداة فعالة، لكنها تضـيف تعقيد، والذي يتعارض مع الضـرورة التقنية الأسـاسـية للبرمجيات لإدارة التعقيد.
- إن الصفوف هي أداتك الأساسية لإدارة التعقيد. لذلك أعطي تصميمها قدر من الاهتمام حسب الحاجة لتحقيق هذا الهدف.

الفصل 7

إجرائيات عالية الكفاءة

المحتويات[1]

مواضيع ذات صلة

- خطوات في بناء الإجرائية: المقطع 9.3
- صفوف ناجحة: الفصل 6
- تقنيات التصميم العامة: الفصل 5
- هندسة البرمجيات: المقطع 3.5

يصف الفصل السادس تفاصيل إنشاء الصفوف، هذا الفصل يركز على الإجرائية، وعلى الهيكلية التي تفرّق بين الإجرائية الجيدة والسيئة، إذا أردت أن تقرأ نسخاً حول تصميم الإجرائية قبل الخوض في التفاصيل المعقدة، أحرص أولاً على أن تقرأ الفصل الخامس "التصميم في البناء" ومن ثم عَد إلى هذا الفصل. بعض الخصائص الهامة للإجرائيات عالية الكفاءة أيضاً مذكورة في الفصل الثامن، "البرمجة الوقائية"، وإن كنت مهتما أكثر بالقراءة حول خطوات إنشاء الإجرائية والصفوف، الفصل التاسع، " معالجة برمجة الشفرة الزائفة" قد يكون المكان الأفضل للبدء.

[1] cc2e.com/0778

قبل القفز لتفاصيل الإجرائية عالية الكفاءة سيكون مفيداً عمل خلاصة نهائية لمصطلحين أساسيين. ما هي "الإجرائية"؟ الإجرائية هي طريقة فردية أو إجراء لا يمكن الاستغناء عنه للوصول الى غاية وحيدة. من الأمثلة عليها: تابع في سي++، طريقة في جافا، تابع أو إجراء فرعي في مايكروسوفت فيجول بيسك، ولبعض استخدامات الماكرو في سي وسي++ يمكن أيضاً أن يقصد فيه إجرائيات. يمكنك تطبيق الكثير من التقنيات لإنشاء إجرائية عالية الكفاءة لهذا التنوع.

ماهي الإجرائية عالية الكفاءة؟ هذا هو أصعب سؤال. ربما الجواب الأسهل هو تبيان ما هو ليس إجرائية عالية الكفاءة وهنا مثال عن إجرائية متدنية الكفاءة:

مثال سي++ لإجرائية منخفضة الكفاءة




```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA
empRec,
double & estimRevenue, double ytdRevenue, int screenX, int screenY,
COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
inputRec.revenue[i] = 0;
inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
}
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
for ( i = 0; i < 12; i++ )
profit[i] = revenue[i] - expense.type1[i];
}
else if ( expenseType == 2 ) {
profit[i] = revenue[i] - expense.type2[i];
}
else if ( expenseType == 3 )
profit[i] = revenue[i] - expense.type3[i];
}
```

ما هو الخطأ في هذه الإجرائية؟ **تنويه**: يجب أن تتمكن من إيجاد عشر أخطاء مختلفة فيه على الأقل، عندما تحضر قائمتك، ألق نظرة على القائمة التالية:

- اسم الإجرائية سيئ ()HandleStuff وهو لا يخبرك شيئاً عما تفعله الإجرائية.

- الإجرائية ليسـت موثقة. (موضـوع التوثيق يمتد لما بعد حدود الإجرائية الخاصـة وهو مناقش في الفصل 32،" ترميز التوثيق الذاتي").
- تخطيط الكتابة (layout) في الإجرائية سـيئ. التنظيم الفيزيائي للترميز في الصـفحة يعطي بعض الاشـارات حول تنظيمها المنطقي، اسـتراتيجيات التخطيط مسـتخدمة كيفما اتفق، بأسـاليب مختلفة في أجزاء مختلفة من الإجرائية. قارن الاسلوبين في expenseType == 2، expenseType == 3. (التخطيط مناقش في الفصل 31 " التخطيط والأسلوب").
- متغيرات دخل الإجرائية، inputRec تغيرت. إن كانت متغيرات دخل فيجب ألا تتعدل قيمتها (وفي لغة سـي++ يجب أن يكون مصـرح عنها const). إذا كانت قيمة المتغير متوقع تعديلها فالمتغير يجب أن لا يسمى inputRec.
- الإجرائية تقرأ وتكتب متغيرات شاملة، أنها تقرأ من corpExpense وتكتب في profit. يجب أن تتصل مع اجرائيات اخرى بشكل مباشر أكثر بدلاً من قراءة وكتابة متغيرات شاملة.
- ليس للإجرائية غاية واحدة. إنها تهيئ بعض المتغيرات وتكتب في قاعدة المعطيات وتقوم ببعض عمليات الجمع ولا واحدة منها تبدو متعلقة بالأخرى بأي طريقة. يجب أن يكون للإجرائية غاية واحدة محددة بوضوح.
- الإجرائية لا تحمي نفسها من المعطيات السيئة. إذا كان crntQtr يساوي الصفر، فإن التعبير ytdRevenue * 4.0 / (double) crntQtr يسبب خطأ القسمة على الصفر.
- تستخدم الإجرائية عدة أرقام سحرية: 100 و4.0 و12 و2 و3. الارقام السحرية مناقشة في القسم 12. 1، "الأرقام بالعموم".
- بعض وسطاء الإجرائية غير مستخدمة: screenX وscreenY لم تذكرا خلال عمل الإجرائية.
- تم تمرير أحد وسـطاء الإجرائية بشـكل خاطئ: prevColor معنونة كوسـيط مرجعي (&) ورغم ذلك لم يتم اسناد قيمة اليها في الاجرائية.
- للإجرائية الكثير جدا من الوسـطاء، الحد الأعلى للوسـطاء القابلة للفهم هو سـبعة، هذه الإجرائية فيها 11، الوسـطاء موضـوعة بطريقة غير قابلة للقراءة لذا أغلب الناس لن يحاولوا فحصـها بدقة أو حتى عدّها.
- وسـطاء الإجرائية مرتبة بشـكل سـيئ وغير موثقة. (ترتيب الوسـطاء مناقش في هذا الفصـل، التوثيق مناقش في الفصل 32)

بعيداً عن الحاسوب نفسه فإن الإجرائية هي أعظم اختراع في علوم الحاسوب[1]. الإجرائية تجعل البرامج أسهل قراءة وفهما بشكل كبير من أي ميزة بأي لغة برمجة، وإنها لجريمة إساءة استخدام هذا الركن من أركان علوم الحاسوب مع شفرة كالذي في المثال الذي عرضناه قبلاً.

الإجرائية أيضاً هي أعظم تقنية تم اختراعها للآن لتوفير المساحة وتحسين الأداء. تخيل كم سيكون كبيراً حجم الشفرة إذا كان عليك أن تعيد الشفرة لكل استدعاء للإجرائية بدلاً من توجيهه إليها. تخيل كم سيكون صعباً إجراء تحسين للأداء في نفس الشفرة المستخدمة في عشرات الأماكن بدلاً من وضعهم جميعاً في إجرائية واحدة، الإجرائية تجعل البرمجة الحديثة ممكنة.

" حسناً " ستقول: " لقد عرفت بأن الإجرائية شيء عظيم، وإنني سأستخدمها في البرمجة دائما، لذا ما الذي تريده مني القيام به من أجلها؟"

أريدك أن تفهم بأن العديد من الأسباب الصحيحة لإنشاء إجرائية موجودة وأنه يوجد العديد من الطرق الصحيحة والطرق الخاطئة لذلك. كطالب علوم الحاسوب غير متخرج، أعتقد بأن السبب الرئيسي لإنشاء إجرائية كان لتجنب تضاعف الشفرة. تقول الكتب الدراسية التي قرأتها بأن الإجرائية كانت جيدة لأن إبطال التضاعف يجعل البرنامج أسهل للتطوير والتصحيح والتوثيق والصيانة. نقطة.

بالإضافة إلى التفاصيل النحوية حول كيفية استخدام الوسطاء والمتغيرات المحلية، والتي كانت امتداد لما تغطيه الكتب الدراسية. لم يكن ذلك كله شرحا جيدا أو كاملا عن نظرية وتطبيق الإجرائية. المقطع التالي يتضمن شرح أفضل لذلك.

1.7 الأسباب الصحيحة لإنشاء الإجرائية:

هنا قائمة بالأسباب الصحيحة لإنشاء إجرائية. الأسباب تتداخل فيما بينها إلى حد ما، وليس المراد منها إجراء إحصائية عددية.



تقليل التعقيد: السبب الأكثر أهمية لإنشاء إجرائية هو تقليل تعقيدات البرنامج. إنشاء إجرائية لإخفاء معلومات وهكذا فلن تكون بحاجة للتفكير فيها. بالتأكيد، ستحتاج للتفكير بها عند كتابة الإجرائية. ولكن بعد كتابتها فإنه يمكنك نسيان التفاصيل واستخدام الإجرائية بدون أي معرفة بآلية عملها الداخلية. أسباب

[1] cc2e.com/0799

اشارة مرجعية الصفوف هي ايضاً تحدي جيد لأعظم اختراع في علوم الحاسوب. للتفاصيل حول استخدام الصفوف بشكل فعال، انظر الفصل 6 "الصفوف الناجحة"

أخرى لإنشاء إجرائيات ـ تقليل حجم الشفرة وتحسين إمكانية الصيانة وتحسين التصحيح ــ هي أيضاً أسباب جيدة، ولكن بدون قوة الإجرائية المجردة؛ فمن المستحيل إدارة البرامج -المقعدة- فكريا.

أحد المؤشرات على الحاجة لاستخلاص اجرائية من إجرائية أخرى هو التعشيش العميق لحلقة داخلية أو العبارات الشرطية. خفف من تعقيد الإجرائية الكبيرة بسحب جزء التعشيش خارجاً ووضعه في إجرائيته الخاصة.

تقديم تجريد متوسط قابل للفهم: وضع قسم من الشفرة في إجرائية مسماة جيداً هو أفضل الطرق لتوثيق أهدافه، بدلاً من قراءة سلسلة من العبارات مثل:

```
if ( node <> NULL ) then
 while ( node.next <> NULL ) do
  node = node.next
  leafName = node.name
 end while
else
 leafName"" =
end if
```

يمكنك قراءة العبارة هكذا:

```
leafName = GetLeafName(node)
```

الإجرائية الجديدة قصيرة جداً لدرجة أن كل ما تحتاجه للتوثيق هو اسم جيد. الاسم يقدم مستوى أعلى من الثمانية أسطر الأصلية المكونة للشفرة، والتي تجعل الشفرة أسهل للفهم وتقلل التعقيد من خلال الإجرائية والتي هي أصلاً مضمنة في الشفرة.

تجنب تكرار الشفرة: بلا شك أكثر الأسباب شعبية لإنشاء إجرائية هو لتفادي تضاعف الشفرة، في الواقع، خلق ترميز متشابه في إجرائيتين يتضمن خطأً في التحليل. اسحب الشفرة المكررة من كلا الإجرائيتين، ضع نسخة شاملة للشفرات المعروفة في صف قاعدة، ومن ثم انقل كلا الإجرائيتين المخصصتين إلى صفوف فرعية. او بدلا من ذلك، نقل الشفرة المشتركة إلى إجرائية خاصة، ومن ثم تدعهما يستدعيان الجزء الذي تم وضعه في إجرائية جديدة. بشفرة في مكان واحد، فإنك توفر المساحة التي ستكون قد استخدمت عبر الشفرة المكررة. التعديل سيكون أسهل لأنك ستحتاج لتعديل الشفرة فقط في مكان واحد. الشفرة ستكون أكثر وثوقية لأنك ستتفحص مكان واحد فقط للتأكد من أن الشفرة صحيحة. التعديل سيكون أكثر وثوقية لأنك ستتجنب إجراء تعديلات مختلفة متتابعة ومهملة تحت افتراض خاطئ بأنك قمت بمطابقته.

دعم التصفيف الفرعي: ستحتاج الى شفرة اقل لتقوم بتحميل زائد لإجرائية قصيرة ومصممة بشكل جيد من الشفرة اللازمة لإجرائية طويلة ومصممة بشكل ضعيف، وستقلل أيضاً فرصة الخطأ في تنفيذ الصفوف الفرعية إن حافظت على الإجرائيات القابلة للتحميل الزائد بسيطة.

إخفاء التسلسل: إنها فكرة جيدة أن تقوم بإخفاء الترتيب الذي يتم فيه معالجة الأحداث. مثلاً: إذا حصل البرنامج بشكل نموذجي على البيانات من المستخدم وبعدها حصل على بيانات مساعدة من ملف ما، فإما الإجرائية التي حصلت على البيانات من المستخدم أو الإجرائية التي حصلت على بيانات الملف سيعتمد على الإجرائية الأخرى كونها تم إنجازها أولاً. مثال آخر للتسلسل ربما يكون موجوداً عندما يكون لديك سطرين من الشفرة التي تقرأ رأس المكدس وتنقص متغيرات رأس المكدس. ضع هذين السطرين من الشفرة في إجرائية الـ popstack() لإخفاء الفرضية حول الترتيب الذي يجب أن تنجز به العمليتين. إخفاء هذه الفرضية سيكون أفضل من حشرها في شفرة من إحدى نهايتي النظام للنهاية الأخرى.

إخفاء عمليات المؤشر: عمليات المؤشر تتجه لتكون صعبة القراءة ومعرضة للخطأ. عبر عزلهم بإجرائيات، تستطيع التركيز على هدف العملية أكثر من التركيز على آلية إدارة المؤشر. أيضاً، إذا أنجزت العملية في مكان واحد فقط، فبإمكانك أن تكون متأكداً أكثر بأن الشفرة صحيحة. إذا وجدت نوع بيانات أفضل من المؤشرات، يمكنك تغيير البرنامج دون تعديل الشفرة التي تستخدم المؤشرات.

تحسين إمكانية النقل: استخدام الإجرائية يعزل المقدرات غير القابلة للنقل، بوضوح، معرفا وعازلا للعمل المستقبلي الخاص بإمكانية النقل. وتشمل المقدرات غير قابلة للنقل الميزات غير للقياسية للغة، تبعية العتاد الصلب، تبعية نظام التشغيل، وهلم جراً.

تبسيط الاختبارات المنطقية المعقدة: نادراً ما يكون فهم تفاصيل الاختبارات المنطقية المعقدة ضرورياً لفهم عمل البرنامج. وضع هكذا اختبارات في تابع يجعل الشفرة أكثر قابلية للقراءة لأن (1) تفاصيل الاختبار بعيدة المنال و (2) الاسم الوصفي للتابع يلخص الغاية من الاختبار.

إعطاء الاختبار لوحده تابع يؤكد أهميته. إنه يشجع بذل جهد زائد لجعل تفاصيل الاختبار قابلة للقراءة ضمن تابعها. النتيجة هي بأن كلاً من التدفق الرئيسي للشفرة والاختبار ذات نفسه أصبح أكثر وضوحاً. تبسيط الاختبار المنطقي هو مثال لتقليل التعقيد، والتي تم مناقشته سابقاً.

تحسين الأداء: يمكنك تحسين الشفرة في مكان واحد بدلاً من عدة أماكن. كون الشفرة في مكان واحد يسهل تحديد المعلومات الأكثر أهمية لإيجاد نقاط الضعف. تمركز الشفرة في إجرائية يعني بأن تحسين واحد سيعود بالنفع على كل الشفرة التي تستخدم الإجرائية. سواء استخدمها بشكل مباشر أو غير مباشر. كون الشفرة في مكان واحد يجعل إعادة كتابة شفرة الإجرائية بخوارزمية أكثر فعالية أو بلغة أكثر كفاءة وسرعة امراً عملياً.

لضمان أن كل الإجرائيات صغيرة؟ لا. مع العديد من الأسباب الجيدة لوضع الشفرة في إجرائية، هذا البند غير ضروري. في الواقع، بعض الأعمال تنجز بشكل أفضل في إجرائية واحدة كبيرة. (الطول الأفضل للإجرائية تم مناقشته في القسم 4.7، "كم من الممكن أن يكون طول الإجرائية؟")

العمليات التي تبدو بسيطة جداً لوضعها في إجرائية

نقطة مفتاحية

أحد أقوى الحواجز الفكرية لإنشاء إجرائية فعالة هو عدم الرغبة بإنشاء إجرائية بسيطة لأهداف بسيطة. إنشاء كامل الإجرائية لتتضمن سطرين أو ثلاثة من الشفرة ربما يبدو مبالغة، ولكن الخبرة تُظهر كم هو مفيد أن تكون الإجرائية صغيرة وجيدة.

الإجرائيات الصغيرة تقدم العديد من الفوائد. إحداها تحسين إمكانية القراءة. في إحدى المرات كان لدي السطر الوحيد التالي من الشفرة في عشرات الأماكن في برنامج:

```
Pseudocode Example of a Calculation
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

وهذا ليس أكثر الأسطر تعقيدا في الشفرة التي ستقرؤونها. ومعظم الناس سيعرفون في نهاية المطاف بأنه يحول القياس بآلية الوحدات إلى القياس بالنقطة. سيرون بان كل دزينة أسطر فعلت نفس الشيء. من الممكن أن تكون أوضح، وهكذا أنشئت إجرائية باسم جيد للقيام بالتحويل في مكان واحد.

```
Pseudocode Example of a Calculation Converted to a Function
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
DeviceUnitsToPoints = deviceUnits*
POINTS_PER_INCH / DeviceUnitsPerInch(())
End Function
```

عندما تم استبدال شفرة جسم الإجرائية بالإجرائية، فإن عشرات الأسطر من الشفرة بدت تقريبا مثل هذا:

```
Pseudocode Example of a Function Call to a Calculation Function
points = DeviceUnitsToPoints( deviceUnits )
```

هذا السطر أكثر قابلية للقراءة – لا بل أقرب للتوثيق الذاتي.

هذا المثال ينوه لسبب آخر لوضع عمليات صغيرة في التابع: العمليات الصغيرة تتجه لان تتحول إلى عمليات أكبر. لم أكن أعلم ذلك عندما كتبت الإجرائية، ولكن تحت شروط محددة وعندما كانت أدوات محددة فعالة، Device-UnitsPerInch() أعادت قيمة (0).وهذا عنى انه عليَّ مراعاة القسمة على الصفر، الأمر الذي أخذ ثلاثة أسطر زيادة في الشفرة.

```
Pseudocode Example of a Calculation That Expands Under Maintenance
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
 if ( DeviceUnitsPerInch() <> 0(
 DeviceUnitsToPoints = deviceUnits*
) POINTS_PER_INCH / DeviceUnitsPerInch(()
 else
  DeviceUnitsToPoints = 0
 end if
End Function
```

إذا كان السطر الأصلي للشفرة لا يزال في عشرات الأماكن، فإن الاختبار سيتكرر عشرات المرات، لما مجموعه 36 سطر من الشفرة. إجرائية بسيطة قللت ال 36 سطر الجديدة إلى 3 أسطر.

ملخص لأسباب إنشاء الإجرائية

هنا لائحة موجزة للأسباب الصحيحة لإنشاء إجرائية:

- تقليل التعقيد
- تقديم تجريد متوسط قابل للفهم.
- تجنب مضاعفة الشفرة.
- دعم التصفيف الفرعي.
- إخفاء الترتيب.
- إخفاء عمليات المؤشر.
- تحسين قابلية النقل.
- تبسيط الاختبارات المنطقية المعقدة.
- تحسين الأداء.

بالإضافة لأن الكثير من أسباب إنشاء الصف هي أسباب جيدة لإنشاء إجرائية:

- عزل التعقيد.
- إخفاء تفاصيل التنفيذ.
- الحد من آثار التغيير.
- إخفاء البيانات الشاملة.
- عمل نقاط مركزية للتحكم.
- تسهيل إعادة استخدام الشفرة.
- اتمام إعادة تصنيع محددة.

2.7 التصميم في مستوى الإجرائية

أدخلت فكرة التماسك بمقالة علمية من قبل واين ستيفنز، جلينفورد مايرز، ولاري قسطنطين (1974). مفاهيم أخرى أكثر حداثة، مثل التجريد والتغليف، تتجه الى انتاج رؤى أكثر في مستوى الصف (وهي، في الواقع، حلت محل التماسك على مستوى الصف)، ولكن التماسك لا يزال موجودا وهو جيد كعمود فقري للتصميم التجريبي على مستوى الإجرائية الافرادي.

بالنسبة للإجرائيات[1]، التماسك يشير إلى مدى تعلق العمليات ببعضها. بعض البرامج تفضل التعبير " قوة ": كم هو قوي ارتباط العمليات ببعضها في الإجرائية؟ إن تابع مثل `Cosine()` متماسك بشكل تام لأن كل الإجرائية مخصصة لإنجاز وظيفة واحدة. إن تابع مثل `CosineAndTan()` له تماسك أقل لأنه يحاول القيام بأكثر من عمل. الهدف هو قيام كل إجرائية بفعل عمل واحد بشكل جيد ولا تفعل أي شيء آخر.

بيانات مثبتة

الفائدة هي وثوقية أعلى. بدراسة ل 450 إجرائية وجدت بأن 50% من الإجرائيات عالية التماسك خالية من الأخطاء، بينما 18% من الإجرائيات بتماسك أقل كانت خالية من الأخطاء (كارد، كرش أغريستي 1986). بدراسة أخرى ل 450 إجرائية مختلفة (والذي هو صدفة غير عادية فقط) وجدت بأن الإجرائيات ذات نسبة "اقتران الى تماسك" اعلى فيها أخطاء أكثر ب 7 مرات من الإجرائيات ذات نسبة "اقتران الى تماسك" ادنى، وكانت 20 مرة أكثر تكلفة لإصلاحها (سليبي وباسيلي 1991).

ترجع النقاشات حول التماسك عادةً إلى عدة مستويات من التماسك. فهم المفاهيم العامة هو أكثر أهمية من تذكر عبارات محددة. استخدم المفاهيم كمساعد في التفكير حول كيفية جعل الإجرائية متماسكة قدر الإمكان.

[1] لمناقشة التماسك بالعموم انظر" التصويب الى تماسك قوي" في القسم 5. 3

التماسك الإجرائي هو أقوى وأفضل أنواع التماسك. يحدث عندما تنجز الإجرائية عملية واحدة وفقط عملية واحدة. أمثلة عن الإجرائية عالية التماسك تتضمن ()sin و GetCustomerName و ()EraseFile و ()CalculateLoanPayment و ()AgeFrom-Birthdate. طبعاً، هذا التقدير للتماسك يفترض بأن الإجرائية تقوم بفعل ما يشير إليه اسمها، إذا كانت تفعل أي شيء آخر فهي أقل تماسكاً ومسماة بشكل سيئ.

العديد من أنواع التماسك وبشكل طبيعي تعتبر أقل مثالية:

- **التماسك المتسلسل:** يكون عندما تتضمن الإجرائية عمليات يجب أن تنفذ بترتيب محدد، وهذا يشارك البيانات من خطوة لخطوة، لكن لا يشكل وظيفة متكاملة عندما تعمل معاً (لعمليات).

مثال عن التماسك المتسلسل هو الإجرائية التي تعطى تاريخ الميلاد وتحسب عمر ووقت التقاعد للموظفين. إذا كانت الإجرائية تحسب العمر وبعدها تستخدم النتيجة لحساب وقت التقاعد للموظف، فإن لها تماسك متسلسل. وإذا كانت الإجرائية تحسب العمر وبعدها تحسب وقت التقاعد بحساب منفصل تماماً وحذا يحدث لاستخدام نفس بيانات تاريخ الميلاد، فإنه تماسك تواصل" فقط.

كيف ستجعل الإجرائية متماسكة وظيفياً؟ عليك إنشاء إجرائيات منفصلة لحساب عمر موظف لديك تاريخ ميلاده وحساب وقت التقاعد لتاريخ ميلاد لديك. إجرائية وقت التقاعد تستطيع استدعاء إجرائية العمر. وكلاهما لديهم تماسك وظيفي. يمكن للإجرائيات الأخرى أن تستدعي أي إجرائية منهما أو كلاهما.

- **تماسك التواصل:** يحدث عندما تستعمل العمليات في الإجرائية نفس البيانات ولا تكون مرتبطة بأي طريقة أخرى. إذا كانت الإجرائية تطبع تقريراً ملخصاً ثم تعيد تهيئة بيانات الملخص المدخلة إليها، فإن للإجرائية تماسك تواصل: كلا العمليتين مرتبطتين فقط بحقيقة أنهم يستخدمون نفس البيانات.

لإعطاء هذه الإجرائية تماسك أفضل، يجب إعادة تهيئة ملخص البيانات بالقرب من مكان انشائها، والتي يجب ألا تكون في إجرائية طباعة التقرير. تقسيم العمليات لإجرائيات فردية. الأولى تطبع التقرير والثانية تعيد تهيئة البيانات بأقرب ما يكون للشفرة التي أنشأت أو عدلت البيانات. استدعاء كلا الإجرائيتين من الإجرائية ذات المستوى الأعلى والتي تسمى أصلاً إجرائية التماسك المتواصلة.

- **التماسك المؤقت:** يحدث عندما تنضم العمليات إلى إجرائية لأنهم يعملون بنفس الوقت. النموذج المثالي سيكون ()Startup،()CompleteNewEmployee، و ()Shutdown. بعض البرامج تعتبر التماسك المؤقت غير مقبول لأنه أحياناً مرتبط بتطبيقات البرامج السيئة كالحصول على خليط من الشفرة في إجرائية ()Startup.

لتجنب هذه المشكلة، فكر بالإجرائية المؤقتة كمنظم للأحداث الأخرى. إجرائية ()Startup، على سبيل المثال، ربما تقرأ ملف التعريف، تهيئ ملف البدء(scratch file) وتقوم بإعداد مدير للذاكرة، وإظهار شاشة أولية. ولجعله فعالاً أكثر، فإن إجرائية التماسك المؤقت تستدعي الإجرائية الأخرى لإنجاز فعاليات

محددة أكثر من إنجاز العمليات مباشرة بنفسها. بهذه الطريقة، سيكون واضحاً بأن غاية الإجرائية هي توزيع الفعاليات أكثر من القيام بهم مباشرة.

هذا المثال يقوي مسألة اختيار الاسم بحيث يصف الإجرائية في المستوى الصحيح من التجريد. بإمكانك اتخاذ القرار بأن تسمي الإجرائية ()ReadConfigFileInitScratchFileEtc، والذي يدل بأن الإجرائية لديها فقط تماسك تصادفي. إذا أسميتها ()startup، بكل الأحوال سيكون من الواضح بأن لها غاية وحيدة وواضح بأن لها تماسك وظيفي.

الأنواع المتبقية من التماسك بشكل عام غير مقبولة. إنها تؤدي لشفرة سيئة التنظيم، وصعبة التصحيح، وصعبة التعديل. إذا كان للإجرائية تماسك سيئ، فإنه من الأفضل بذل الجهد لإعادة كتابتها للحصول على تماسك أفضل أكثر من التحقيق في تحليل بالغ الدقة للمشكلة، معرفة ما يجب تجنبه يمكن أن يكون مفيداً، بكل الأحوال، لدينا هنا أنواع غير مقبولة من التماسك:

- **التماسك الإجرائي:** يحدث عندما تتم العمليات في الإجرائية بترتيب محدد. كمثال الإجرائية التي تحصل على أسماء الموظفين ومن ثم عناوينهم وبعدها رقم الهاتف. إن ترتيب هذه العمليات مهم فقط لأنها تماثل الترتيب الذي سيطلب من المستخدم للبيانات على شاشة الإدخال. إجرائية أخرى تأخذ بقية بيانات الموظف. الإجرائية لها تماسك إجرائي لأنها تضع مجموعة من العمليات بترتيب محدد ولا تحتاج العمليات لان تتحد لأي سبب آخر.

  لتحقيق تماسك أفضل، ضع العمليات المنفصلة في إجرائيات. تأكد أن للإجرائية المستدعاة عمل واحد متكامل: ()GetEmployee بدلا من ()GetFirstPartOfEmployeeData. ربما ستحتاج لتعديل الإجرائيات التي تأخذ بقية البيانات أيضاً. إنه من الشائع تعديل إجرائيتين أصليتين أو أكثر قبل تحقيق التماسك الوظيفي في أي منهم.

- **التماسك المنطقي:** وهو يحدث عندما تقحم العديد من العمليات في نفس الإجرائية ويتم اختيار إحدى العمليات عبر علامة تحكم تمرر فيها. تسمى التماسك المنطقي لان سير التحكم أو " منطق " الإجرائية هو الشيء الوحيد الذي يربط العمليات ببعضها، أنهم جميعاً ضمن عبارة if كبيرة أو عبارة case معاً. ليس لأن العمليات مرتبطة منطقياً بأي معنى أخر. معتبرين أن تعريف خاصية التماسك المنطقي هي أن العمليات غير مرتبطة ببعضها، ربما الاسم الأفضل لها "التماسك غير المنطقي".

  أحد الأمثلة سيكون إجرائية ()inputall التي تدخل أسماء الزبائن، معلومات بطاقة الدوام للموظف، أو بيانات الجرد معتمدة على إشارة تمرر للإجرائية. أمثلة أخرى ستكون ()ComputeAll و()EditAll و ()printall و ()saveall. المشكلة الأساسية مع هكذا إجرائيات أنه لا ينبغي ان تحتاج لتمرير إشارة للتحكم بمعالجة إجرائية أخرى. بدلاً من الحصول على إجرائية تقوم بواحدة من ثلاثة عمليات استثنائية، معتمدة على إشارة ممررة لها، إنه من الأنظف ان يكون لدينا ثلاثة إجرائيات، كل واحدة

منهم تقوم بعملية استثنائية. إذا استخدمت العمليات بعض التعليمات البرمجية نفسها أو تشاركت البيانات، فيجب نقل الشفرة إلى داخل إجرائية ذات مستوى أقل كما يجب تحزيم الإجرائيات في صف.

من المناسب عادة، على كل الأحوال[1]، إنشاء إجرائية متماسكة منطقياً إذا كانت شفرتها البرمجية تتكون فقط من سلسلة من عبارات IF أو CASE وتُستدعى لإجرائيات أخرى. في مثل هذه الحالة، إذا كانت وظيفة الإجرائية فقط إعطاء الأوامر ولا تقوم بأي معالجة بنفسها، فإنه عادة يعتبر تصميم جيد. المصطلح التقني لهذا النوع من الإجرائية هو "معالج الأحداث". غالبا ما يستخدم معالج الأحداث في بيئات تفاعلية مثل أبل ماكنتوش ومايكروسوفت ويندوز وبيئات واجهة المستخدم الرسومية الأخرى.

- **التماسك التصادفي**: وهو يحدث عندما لا يكون للعمليات في الإجرائية علاقات مميزة فيما بينها. الأسماء الجيدة الأخرى هي "لا تماسك" أو "تماسك مشوش". الإجرائية منخفضة الجودة بلغة سي++ في بداية هذا الفصل لها تماسك متطابق. ومن الصعب تحويل التماسك المتطابق إلى أي نوع آخر أفضل من أنواع التماسك ـ عادة تحتاج لإعادة تصميم أعمق وإعادة تحقيق.



ولا واحد من هذه الحدود سحري أو مقدس. تعَلم الأفكار بدلا من المصطلحات. إنه تقريباً من الممكن دائماً كتابة إجرائيات مع تماسك وظيفي، لذا عليك تركيز انتباهك على التماسك الوظيفي لتحقيق أقصى فائدة.

3 .7 الأسماء الجيدة للإجرائيات[2]

الاسم الجيد للإجرائية يصف بوضوح كل ما تقوم به هذه الإجرائية. هنا الإرشادات لإنشاء أسماء إجرائيات فعّالة:

وصف كل ما يفعله الإجراء: في اسم الإجرائية، صف كل المخرجات والآثار الجانبية. إذا كان تقرير الإجرائية يحسب المجموع ويفتح ملف خرج، فإن اسم `Compute-ReportTotals()` ليس ملائماً للإجرائية.

و `ComputeReportTotalsAndOpen-OutputFile()` اسم ملائم ولكنه طويل جداً وساذج. إذا كان لديك إجرائيات بآثار جانبية سيكون لديك العديد من الأسماء الطويلة والساذجة. والعلاج ليس باستخدام أسماء أقل وصفاً للإجرائية، العلاج هو ان تبرمج بحيث تسبب حدوث أشياء مباشرةً أكثر من الآثار الجانبية.

[1] **إشارة مرجعية** بالرغم من أنه يمكن أن يكون للإجرائية تماسك أفضل، تظهر قضية تصميم عالي المستوى هي إذا كان ينبغي ان يستخدم النظام عبارة case بدلاً من تعدد الأشكال. للمزيد عن هذه القضية انظر " الاستبدال المشروط بتعدد الاشكال. (بشكل خاص عبارات case المتكررة)" في القسم 24. 3

[2] للتفاصيل عن تسمية المتغيرات انظر القسم 11. " قوة أسماء المتغيرات"

تجنب الأفعال الغامضة وغير المركزة: بعض الأفعال مرنة، وممتدة لتغطي اي معنى تقريباً. أسماء الإجرائيات مثل: `HandleCalculation()` و `PerformServices()` و `OutputUser()` و `ProcessInput()` و `DealWithOutput()` لا تخبرك بما يفعل الإجراء. غالباً هذه الأسماء تخبرك بأن الإجرائية تقوم بفعل شيء ما له علاقة بالعمليات الحسابية، والخدمات والمستخدمون والدخل والخرج. الاستثناء سيكون عندما يستخدم الفعل " يتعامل" بالمعنى التقني للتعامل مع حدث ما.



أحياناً المشكلة الوحيدة مع الإجراءات أن اسمها غير مركز، الإجرائية ذات نفسها في الواقع تكون مصممة بشكل جيد. إذا استبدلت اسم الإجرائية `HandleOutput()` بالاسم `FormatAndPrintOutput()` سيكون لديك فكرة جيدة وجميلة عما تفعله الإجرائية.

في حالات أخرى، يكون الفعل غامض لأن العمليات المنجزة من قبل الإجرائية تكون غامضة. وتعاني الإجرائية من ضعف الغاية، والاسم الضعيف هو إشارة إنذار. إن كانت تلك هي الحالة، فالحل الأفضل هو إعادة هيكلة الإجرائية وأي إجرائية أخرى مرتبطة بها بحيث يكون لهم جميعاً غايات أقوى وأسماء أقوى بحيث تصفهم بشكل دقيق.



لا تفرق بين أسماء الإجرائية فقط بالأرقام: أحد المطورين كتب كل شفرته بإجراء واحد كبير، بعدها أخذ كل 15 سطراً وأنشأ إجرائيات سماها الجزء 1 والجزء 2 وهكذا. بعد ذلك، أنشأ إجرائية عالية المستوى بحيث تُسمي كل جزء. هذه الطريقة بإنشاء وتسمية الإجرائية رديئة جداً (ونادرة، آمل ذلك). لكن المبرمجين أحياناً يستخدمون الأرقام لتفريق الإجرائية بأسماء مثل: `OutputUser` و`utputUser1`، و`OutputUser2`. الترقيم في نهاية هذه الأسماء لا يزودنا بدلائل عن الأفكار التجريدية المختلفة التي تشرحها الإجرائية. والإجرائية تكون قد سَميت بشكل سيئ.

اجعل أسماء الإجرائيات طويلة بقدر الضرورة: تبين الأبحاث بأن معدل الطول الأمثل لأسماء المتغيرات هو من 9 إلى 15 محرفاً. الإجرائية تتجه لأن تكون أكثر تعقيداً من الأسماء المتغيرة، والأسماء الجيدة لها تتجه لأن تكون أطول. من جهة أخرى، أسماء الإجرائية تكون غالباً مرتبطة بأسماء الأشياء، والتي أساساً تزودنا بجزء من الاسم مجاناً. عموماً، التشديد عند إنشاء اسم الإجرائية سيجعل الاسم واضح قدر الإمكان، مما يعني بأنه عليك جعل أسمائها طويلة أو قصيرة حسب الحاجة لجعلها مفهومة.

لتسمية تابع، استخدم وصف للقيمة التي يعيدها[1]: التابع يعيد قيمة، ويجب أن يسمى بحسب القيمة التي يعيدها. مثلاً، cos() و customerId.Next() و printer.IsReady() و pen.CurrentColor() جميعها أسماء توابع جيدة والتي تشير تماماً لما تعيده التوابع.

لتسمية إجراء، استخدم أفعالا قوية متبوعة بكائن: الإجراء ذو التماسك الوظيفي عادة ينجز عملية على كائن. الاسم يجب أن يعكس ما يفعله الإجراء، والعملية على كائن توحي باسم هو فعل-و-كائن. مثل: PrintDocument() و CalcMonthlyRevenues() و CheckOrderInfo() و RepaginateDocument() هي نماذج لأسماء جيدة للإجراء.

في لغات البرمجة غرضية التوجه، لن تحتاج لتضمين اسم الكائن في اسم الإجراء لأن الكائن نفسه مضمن في الاستدعاء. انت تستدعي الإجرائيات بعبارات مثل: document.Print() و orderInfo.Check() و monthlyRevenues.Calc(). بينما أسماء مثل: document.PrintDocument() هي زائدة وممكن أن تصبح خاطئة عندما تُحمل عبر صفوف مشتقة. إذا كان Check صف مشتق من Document و check.Print() يبدو بوضوح أنه يطبع قيمة الـ check، حيث: check.PrintDocument() يعطي انطباعاً بانه يطبع سجل الحوالات أو السجل الشهري. ولا يعطي انطباعاً بأنه يطبع الحوالة.

استخدم المتضادات بدقة[2]: استخدام تسميات اصطلاحية للمتضادات يساعد على التناغم، الذي يساعد على إمكانية القراءة، الأزواج المتضادة مثل first/last مفهومة بشكل عام. الأزواج المتضادة مثل FileOpen() و _ close() ليست متناسقة ومربكة. هنا قائمة ببعض المتضادات المعروفة:

Add/remove إضافة/ إزالة	increment/decrement زيادة / نقصان	open/close فتح / إغلاق
begin/end بداية /نهاية	insert/delete إدخال / مسح	show/hide إظهار / إخفاء
create/destroy إنشاء/تدمير	lock/unlock قفل / فتح	source/target مصدر / هدف
first/last أول / آخر	min/max صغرى / قصوى	start/stop بدء / توقف
get/put أخذ/وضع	next/previous التالي / السابق	up/down صعود /نزول
get/set اقرأ قيمة / عين قيمة	old/new قديم / جديد	

[1] **إشارة مرجعية** للتمييز بين الإجراءات والتوابع، انظر القسم 7 .6، " شروط خاصة في استخدام التوابع" لاحقاً في هذا الفصل.

[2] **إشارة مرجعية** لقائمة مشابهة من المتضادات في اسماء المتغيرات انظر "متضادات مشهورة في أسماء المتغيرات" في القسم 11 .1

أسس اتفاقيات للعمليات الشائعة: في بعض الأنظمة، من المهم التمييز بين الأنواع المختلفة من العمليات. تسمية الاتفاقية هي غالباً الطريقة الأسهل والأكثر وثوقية للإشارة لهذه الفروق.

حددت الشفرة في أحد مشاريعي كل غرض بمعرف وحيد. نحن أهملنا تأسيس اتفاقية لتسمية الإجرائية التي سوف تعيد معرف الغرض، فحصلنا على أسماء إجرائيات كهذه:

```
employee.id.Get()
dependent.GetId()
supervisor()
candidate.id()
```

كشف الصف Employee كائن هويته الخاصة، والذي كشف بدوره إجرائيته ()Get. الصف Dependent كشف الإجرائية ()GetId. الصف Supervisor جعل الهوية قيمته المرتجعة الافتراضية.[1] الصف Candidate استخدام حقيقة أن القيمة الافتراضية المرتجعة لكائن الهوية هو الهوية، وكشف كائن الهوية. في منتصف المشروع، لم يتمكن أحد من التذكر أي من هذه الإجراءات كان من المفترض أن يُستخدم وعلى أي كائن، في ذلك الوقت الكثير من الشفرة البرمجية كان قد كتب للعودة وجعل كل شي متناسق. وبناءً على ذلك، على كل شخص في الفريق أن يخصص بدون حاجة كمية من "المادة الرمادية" لتذكر التفاصيل غير الهامة لأي عبارة كانت قد استخدمت على اي صف لاستعادة الهوية (id). التسمية الاصطلاحية لاستعادة الهويات يجب أن تبطل هذا الإزعاج.

4.7 كم من الممكن أن يكون طول الإجرائية؟

بطريقهم إلى أمريكا، تجادل الرواد الأوائل حول أفضل أقصى طول للإجرائية. بعد النقاش حولها طيلة الرحلة، وصلوا إلى بليموث روك تبدؤوا بصياغة اتفاقية Mayflower (مايفلور). حتى الآن لم ينتهوا من سؤال من سؤال أقصى طول، وحيث انهم لن يستطيعوا النزول من السفينة حتى يوقعوا الاتفاقية، فقد استسلموا ولم يضمنوا السؤال الى لاتفاقية. النتيجة نقاش لا ينتهي حول كم من الممكن ان يكون طول الإجرائية.

نظرياً أفضل أقصى طول هو غالباً شاشة واحدة أو صفحة أو صفحتين من قوائم البرنامج، بشكل تقريبي 50 إلى 150 سطراً. بهذا المجال، شركة آي بي أم IBM حددت مرة إجراءاتها بـ 50 سطراً، وشركة تي آر دبليو TRW حددتهم بصفحتين (مكابي 1976). البرامج الحديثة تتجه لامتلاك أحجام من الإجرائيات القصيرة جداً المختلطة ضمن بعض الإجرائيات الأطول. الإجرائية الطويلة بعيدة عن الانقراض، على كل حال، قبل فترة قصيرة من الانتهاء من هذا الكتاب، قمت بزيارة موقعين للزبائن لمدة شهر. كان المبرمجين في أحد المواقع

[1] **أضف الى معلوماتك** في علم التشريح المادة الرمادية هي التي تكون قشرة الدماغ، والتي تعتبر مركز الادراك والارادة

يتصارعون مع إجرائية والتي كانت بطول حوالي /4000/ سطر من الشفرة، والمبرمجين في الموقع الآخر كانوا يحاولون ترويض اجرائية بطول أكثر من /12.000/ سطراً.

جبل من الأبحاث حول طول الإجرائية قد تكدّس على مدى السنوات، بعضها قابل للتطبيق على البرامج الحديثة وبعضها غير قابل:

بيانات مثبتة

- دراسة أجراها باسليلي وبيركون وجدت بأن حجم الإجرائية يتناسب عكساً مع الأخطاء: كلما زاد حجم الإجرائية (حتى 200سطر من الشفرة)، فإن عدد الأخطاء في كل سطر من الشفرة ينقص (باسيلي وبيركون 1984).
- دراسة أخرى وجدت بأن حجم الإجرائية لم يكن مرتبطاً بالأخطاء، على الرغم من أن التعقيدات الهيكلية وكمية البيانات كانت مرتبطة بالأخطاء (شين وآخرون 1985).
- دراسة عام 1986 وجدت بان الإجرائيات الصغيرة (32 سطر من الشفرة أو أقل) لم تكن مرتبطة بتكلفة أقل أو بنسبة الخطأ (كارد، تشرش، و أعريستي 1986، كارد وغلاس 1990) الأدلة اقترحت بان الإجرائية الأكبر (65 سطر من الشفرة أو أكثر) هي الأقل عناءً للتطوير لكل سطر من الشفرة.
- دراسة تجريبية ل 450 إجرائية وجدت بأن الإجرائية الصغيرة (هؤلاء بأقل من 143 عبارة مصدرية، متضمنةً التعليقات) لها 23% أخطاء أكثر بكل سطر من الشفرة من الإجرائية الأكبر بالمقابل كانت 2.4 مرات أقل كلفة للإصلاح من الإجرائية الأكبر (سيلبي و باسيلي 1991).
- دراسة أخرى وجدت بأن الشفرة تحتاج لتغيير بأقل مدى ممكن عندما يكون معدل الإجرائية من 100 إلى 150 سطر من الشفرة (لايند و فايرفان 1989).
- دراسة لشركة آي بي إم وجدت بأن معظم الإجرائية المعرضة للخطأ كانت تلك التي هي أكبر من 500 سطر من الشفرة. ما بعد الـ 500 سطر معدل الخطأ يميل لأن يكون متناسباً مع حجم الإجرائية (جونز 1986).

أين يترك كل ذلك السؤال عن طول الإجرائية في البرامج غرضية التوجه؟

النسبة الكبرى من الإجرائيات في البرامج غرضية التوجه ستكون إجرائيات وصول، والتي ستكون قصيرة جداً. من فترة لأخرى، خوارزمية معقدة ستؤدي إلى إجرائية أطول، وبهذه الظروف، يجب أن يسمح للإجرائية بالنمو أساساً حتى 100-200 سطر. (السطر هو ليس تعليق، وليس فراغ، بل سطر من شفرة المصدر).

عقد من البراهين تقول بأن الإجرائية بأطوال كهذه ليست معرضة للخطأ بعد الآن أكثر من الإجرائية الأقصر. السماح بقضايا كتماسك الإجرائية، عمق التعشيش، عدد المتحولات، عدد نقاط القرار، عدد التعليقات المطلوبة

لشرح الإجرائية والشروط المعقدة الأخرى المتعلقة بها تحدد طول الإجرائية أكثر من فرض قيود طولية بحد ذاتها.

أحدهم يقول: إذا أردت أن تكتب إجرائية أطول من 200 سطر، كن حذراً، ولا واحدة من الدراسات التي قدمت تقريرها زادت الكلفة، أو أنقصت نسبة الخطأ، أو كليهما بإجرائيات أكبر تمت مناقشتها بين أحجام أكبر من 200 سطر، وأنت لا بد ستصطدم بأعلى حد لقابلية الفهم إذا تجاوزت 200 سطراً من الشفرة.

7. 5 كيفية استخدام وسطاء الإجرائية

بيانات مثبتة

الواجهات بين الإجرائيات هي من أكثر مناطق البرنامج عرضة للخطأ. إحدى أكثر الدراسات التي استشهد بها باسيلي وبيركوني (984) وجدت بأن 39% من جميع الأخطاء كانت أخطاء واجهات داخلية – أخطاء بالاتصال بين الإجرائيات. هنا بعض المبادئ لتقليل أخطاء الواجهات:

ضع الوسطاء بترتيب إدخال – تعديل – إخراج[1]: بدلاً من ترتيب الوسطاء عشوائياً أو هجائياً، ضع الوسطاء التي هي إدخال فقط أولاً، إدخال – و – إخراج ثانياً، وإخراج فقط ثالثاً. هذا الترتيب يدل على التتالي للعمليات التي تحدث ضمن بيانات الإدخال للإجرائية، تغييرها، وإعادة النتيجة. هنا أمثلة لوسطاء مرتبة في Ada

مثال Adaلوسطاء بترتيب إدخال – تعديل – إخراج:

```
procedure InvertMatrix(
        originalMatrix: in Matrix;
    resultMatrix: out Matrix
);
...
 procedure ChangeSentenceCase(
     desiredCase: in StringCase;
     sentence: in out Sentence
);
...
procedure PrintPageNumber(
    pageNumber: in Integer;
    status: out StatusType
);
```

Ada يستخدم كلمات مفتاحية دخل وخرج لجعل وسطاء الدخل

[1] **اشارة مرجعية** لتفاصيل عن توثيق وسطاء الإجرائيات انظر " انتقاد الإجرائيات" في القسم 32 .5.

لتفاصيل عن تهيئة الوسطاء انظر القسم 31 .7 "رسم الاجرائيات ".

هذا الترتيب الاصطلاحي يتعارض مع اصطلاح مكتبة سي بوضع الوسطاء المعدلة أولاً. مصطلحات الإدخال – تعديل – إخراج تعطيني احساس أكثر، ولكن إذا رتبت الوسطاء بشكل ثابت بطريقة ما، فإنك سوف تبقى تقدم خدمة لقرّاء شفرتك البرمجية.

خذ بعين الاعتبار إنشاء كلماتك المفتاحية للدخل والخرج: لغات البرمجة الحديثة الأخرى لا تدعم كلمات مفتاحية للدخل والخرج كما يفعل Ada. في هذه اللغات، ربما لا تزال قادراً لاستعمال المعالج التمهيدي لإنشاء كلماتك المفتاحية الخاصة.

مثال سي ++ لتعريف كلمات الدخل والخرج المفتاحية الخاصة بك

```
#define IN
#define OUT
void InvertMatrix(
   IN Matrix originalMatrix،
   OUT Matrix *resultMatrix
);
...
void ChangeSentenceCase(
   IN StringCase desiredCase،
   IN OUT Sentence *sentenceToEdit
);
...
void PrintPageNumber(
   IN int pageNumber،
   OUT StatusType &status
);
```

في هذه الحالة، ماكرو الكلمات المفتاحية الدخل والخرج استخدمت لغايات التوثيق. لجعل قيمة الوسيط قابلة للتغيير بالإجرائية المُستدعاة. لايزال على الوسيط أن يُمرر كمؤشر أو كوسيط مرجعي.

قبل تبني هذه التقنية، خُذ بعين الاعتبار ثنائي العيوب الهامة التالية. يوسّع تعريف الكلمات المفتاحية الخاصة بك للدخل والخرج لغة البرمجة سي++ بطريقة غير مألوفة لمعظم الناس الذين سيقرؤون شفرتك البرمجية. إذا وسِعت اللغة بهذه الطريقة، تأكد من قيامك بهذا بشكل متّسق، ويُفضل أن يكون ذلك على نطاق المشروع كاملاً. المحدودية أو العيب الثاني هو أن الكلمات المفتاحية للدخل والخرج لن تكون ممكنة التنفيذ عبر المترجم البرمجي، مما يعني بأنه من الممكن تسمية الوسيط كدخل، ومن ثم تعديله داخل الإجرائية بأية طريقة كانت. هذا من الممكن أن يُضلل قارئ شفرتك البرمجية بافتراض أن شفرتك صحيحة بينما هي ليست كذلك. سيكون استخدام الكلمة المفتاحية const في لغة البرمجة سي++ بشكل طبيعي الوسيلة المفضلة لتعريف وسطاء الدخل فقط.

إذا كانت مجموعة من الإجرائيات تستخدم وسطاء متشابهة، ضع هذه الوسطاء المتشابهة في ترتيب مُتناسق. يمكن أن يكون ترتيب وسطاء الإجرائية مُساعد على التذكر، ويمكن أن يجعل الترتيب غير المتناسق الوسطاء صعبة التذكر. مثلاً، في لغة البرمجة سي، الإجرائية `fprintf()` هي نفس الإجرائية `printf()` باستثناء أنها تُضيف ملف كمعامل أول. و الإجرائية `fputs()` هي نفس الإجرائية `puts()` باستثناء أنها تُضيف ملف كمعامل أخير. يجعل هذا الفارق المزعج التافه وسطاء هذه الإجرائيات صعبة التذكر، أكثر مما ينبغي أن تكون عليه.

بالمقابل، تأخذ الإجرائية `strncpy()` معاملات السلسلة المحرفية المُستهدفة، السلسلة المحرفية الهدف، العدد الأعظمي من البايتات، في هذا الترتيب، و تأخذ الإجرائية `memcpy()` نفس المعاملات بنفس الترتيب. يُساعد التشابه بين الإجرائيتين على تذكر الوسطاء في أي من الإجرائيتين.

بيانات مثبتة

استخدم جميع الوسطاء. إذا كنت تُمرر وسيط إلى إجرائية، فاستخدم هذا الوسيط. إذا لم تستخدم هذا الوسيط، فأزله من واجهة الإجرائية. حيث ترتبط الوسطاء غير المستخدمة بنسبة زيادة الأخطاء. في إحدى الدراسات، 49% من الإجرائيات، التي لا يوجد فيها متغيرات غير مستخدمة، ليس فيها أخطاء، وفقط 17% إلى 29% من الإجرائيات، التي فيها أكثر من متغير غير مستخدم، ليس فيها أخطاء (كارد، تشيرش، و اغريستي 1986).

قاعدة إزالة الوسطاء غير المستخدمة هذه لها استثناء واحد. إذا كنت تقوم بتجميع قسم من برنامجك بشكل مشروط، فربما تجمّع أجزاء من إجرائية، تستخدم وسيط معين[1]. كُن قلق بالنسبة لهذه الممارسة، ولكن ان كنت مقتنعاً بأنها ستعمل، هذا جيد أيضاً. بالعموم، إذا كان لديك سبب جيد لعدم استخدام وسيط، أمضي قدماً واترك الوسيط مكانه. أما إذا لم يكن لديك سبب جيد، عندها قُم بحذف النص البرمجي.

ضع الحالة أو متغيرات الخطأ أخيراً. حسب الاتفاقية، تُوضع متغيرات الحالة والمتغيرات التي تشير لخطأ قد حدث، في نهاية قائمة الوسطاء. إنهم حالة طارئة للغاية الرئيسية للإجرائية، وهم وسطاء خرج فقط، لذلك فهي اتفاقية مقنعة.

لا تستخدم وسطاء الإجرائية كمتغيرات عاملة. إنه من الخطورة استخدام وسطاء تم تمريرها إلى إجرائية كمتغيرات عاملة. استخدم متغيرات محلية بدلاً عنها. مثلاً في جزء جافا التالي، تم استخدام المتغير `inputVal` بشكل غير صحيح لتخزين نتائج المتوسط الحسابي:

[1] المقصود بالتجميع هنا ترجمة الشفرة الى لغة الالة

مثال بلغة البرمجة جافا لاستخدام غير مناسب لوسطاء الدخل

```
int Sample( int inputVal ) {
inputVal = inputVal * CurrentMultiplier( inputVal );
inputVal = inputVal + CurrentAdder( inputVal );
...
return inputVal;
}
```

في هذه النقطة، لم يعد يحتوي inputVal القيمة التي سبق وأدخلت

في هذا الجزء من النص البرمجي المتغير inputVal مضلل لأنه في وقت تنفيذ السطر الأخير لم يعد يحتوي قيمة inputVal الأصلية (القيمة المدخلة)؛ إنما يحتوي قيمة محسوبة على أساس جزء من قيمة الدخل، وبهذه الطريقة يكون قد فقد معنى تسميته. وإذا احتجت لاحقاً لتعديل الإجرائية لاستخدام قيمة الدخل الأصلية في بعض الأماكن الأخرى، من المحتمل أن تستخدم المتغير inputVal، وتفترض بأنه يحوي قيمة الدخل الأصلية بينما في الواقع هو ليس كذلك.

كيف تحلّ هذه المشكلة؟ هل تستطيع حلّها بإعادة تسمية المتغير inputVal؟ ربما لا. يمكنك تسمية المتغير بشيء ما مثل workingVal ولكن هذا حل غير كامل، لان الاسم يعجز عن الإشارة بأن قيمة المتغير الأصلية تأتي من خارج الإجرائية. بإمكانك تسمية المتغير تسمية سخيفة مثل inputValThatBecomesWorkingVal أو تستسلم بشكل كامل وتسميه X أو Val، ولكن جميع هذه الطرق ضعيفة.

طريقة أفضل هي تجنب المشاكل الحالية والمستقبلية باستخدام متغيرات عاملة بشكل واضح. يوضح النص البرمجي التالي هذه التقنية:

مثال بلغة البرمجة جافا للاستخدام الجيد لوسطاء الدخل

```
int Sample( int inputVal ) {
int workingVal = inputVal;
workingVal = workingVal * CurrentMultiplier( workingVal );
workingVal = workingVal + CurrentAdder( workingVal );
...

...
return workingVal;
}
```

إذا احتجت استخدام القيمة الأصلية للمتغير inputVal هنا أو في أي مكان آخر. فإنه لايزال ممكناً.

يوضح إدخال المتغير الجديد workingVal دور المتغير inputVal ويلغي فرصة الخطأ باستخدام inputVal في الوقت الخطأ. (لا تأخذ هذا المنطق كتبرئة من التسمية الحرفية للمتغير inputVal أو المتغير workingVal. بالعموم، إن inputVal و workingVal هي أسماء سيئة للمتغيرات، وهذه الأسماء استخدِمت في هذا المثال لجعل أدوار المتغيرات واضحة.)

يؤكد إسناد قيمة الدخل لمتغير عامل على المكان الذي يأتي منه المتغير. إنه يلغي إمكانية تغيير متغير من قائمة الوسطاء بالصدفة. في لغة البرمجة سي ++، يمكن تنفيذ هذه الممارسة عن طريق المترجم البرمجي باستخدام الكلمة المفتاحية const. إذا أشرت إلى الوسيط باستخدام const، فإنه لن يُسمح لك بتعديل قيمته داخل الإجرائية.

وثّق افتراضيات الواجهة حول الوسطاء[1]. إذا افترضت بأن البيانات التي تُمرر إلى الإجرائية تملك خصائص محددة، وثّق هذه الافتراضيات كما أنشأتها. ليس توثّيق افتراضاتك تضييعا للجهد في كلا الإجرائية نفسها وفي المكان الذي تُستدعى منه الإجرائية. لا تنتظر حتى تنتهي من كتابة الإجرائية، ومن ثم العودة وكتابة التعليقات – لن تتذكر عندها كل افتراضاتك. وحتى أفضل من التعليق على الافتراضات، استخدم التأكيدات لوضع الافتراضات داخل الشفرة.

ما أنواع افتراضات الواجهة حول الوسطاء التي يجب أن توثقها؟

- فيما إذا كانت الوسطاء هي وسطاء دخل فقط، أو وسطاء معدّلة، أو وسطاء خرج فقط
- واحدة الوسطاء الرقمية (بوصة، قدم، متر، وهكذا)
- معاني حالة الشفرة وقيم الأخطاء إذا لم تكن تستخدم الأنواع التعدادية
- مجال القيم المتوقّعة
- القيم الخاصة التي يجب ألا تظهر أبداً

بيانات مثبتة

حدّد عدد وسطاء الإجرائية بحوالي سبعة. سبعة هو رقم سحري لإدراك الناس. لقد وَجدت الأبحاث النفسية أن الناس بالعموم لا يستطيعون المحافظة على تتبع أكثر من حوالي سبعة قطع من المعلومات بالمرة الواحدة (ميلر 1956). لقد تم تطبيق هذا الاكتشاف على عدد ضخم من المتدربين، ولقد وُجد بأنه لا يستطيع معظم الناس المحافظة على تتبع أكثر من حوالي سبعة وسطاء للإجرائية في المرة الواحدة.

في الممارسة العملية، يعتمد مدى امكانية الحد من عدد الوسطاء، على كيفية معالجة لغتك لأنواع البيانات المعقدة. إذا كان برنامجك مكتوب بلغة برمجة حديثة تدعم البيانات الإنشائية، يمكنك تمرير نوع بيانات مُركب يحتوي على 13 حقلاً ويمكنك أن تتعامل معه كقطعة ذهنية واحدة من البيانات. أما إذا كان برنامجك مكتوب بلغة أكثر بدائيةً، فربما ستحتاج لتمرير كل الـ 13 حقلاً كلاً على حدى.

[1] **إشارة مرجعية:** لمزيد من التفاصيل حول افتراضات الواجهة، انظر مقدمة الفصل 8 "البرمجة الوقائية". ولمزيد من التفاصيل حول التوثيق، انظر الفصل 32 "الشفرة ذاتية التوثيق".

إذا وجدت نفسك تمرر باستمرار أكثر من بضعة معاملات، هذا يعني أن الاقتران بين الإجرائيات قوي جداً.[1] صمّم الإجرائية أو مجموعة الإجرائيات بطريقة يتم فيها تقليل الاقتران. إذا كنت تمرر نفس البيانات للكثير من الإجرائيات المختلفة، جمّع هذه الإجرائيات في صف وتعامل مع البيانات المستخدمة بشكل مُتكرر على أساس بيانات صف.

خذ بعين الاعتبار اصطلاح تسمية الدخل، والتعديل والخرج للوسطاء. إذا وجدت بأنه من المهم التمييز بين وسطاء الدخل والتعديل، والخرج، أسس اصطلاح للتسمية يعّرفهم. بإمكانك أن تبدأ التسميات بالبادئات التالية (i_،m_،o_)، أو يمكنك أن تبدأهم بـ Input_، Modify_، و Output_.

مرّر المتغيرات أو الكائنات التي تحتاجها الإجرائية للحفاظ على واجهتها المُجردة. يوجد مدرستان تتنافسان على التفكير بكيفية تمرير عناصر كائن إلى الإجرائية. افترض بأنه لديك كائن يعرض البيانات من خلال 10 إجرائيات وصولية والإجرائية التي يتم استدعائها تحتاج ثلاثة مكونات من هذه البيانات للقيام بعملها.

يُجادل أنصار المدرسة الاولى من التفكير، بأنه يجب أن تُمرر فقط العناصر الثلاثة المحددة التي تحتاجها الإجرائية. يجادلون بأن القيام بذلك سيحافظ على الحد الادنى من الاتصال بين الإجرائيات؛ ويقلل الاقتران؛ ويجعل الإجرائيات أسهل للفهم ولإعادة الاستعمال وهكذا. يقولون بأن تمرير كامل الكائن إلى إجرائية ينتهك مبدأ التغليف من خلال احتمال كشف كامل إجرائيات الوصول العشرة على الإجرائية التي تم استدعائها.

يُجادل أنصار المدرسة الثانية بأنه يجب تمرير كامل الكائن. هم يجادلون بأن الواجهة يمكن أن تبقى أكثر ثباتاً إذا كان للإجرائية المُستدعاة المرونة لاستخدام عناصر إضافية من الكائن بدون أن تتغير واجهة الإجرائية. ويجادلون بأن تمرير ثلاث عناصر محددة ينتهك مبدأ التغليف عبر عرض عناصر البيانات المحددة المُستخدمة من قبل الإجرائية.

أعتقد بأن كلا القاعدتين مبسطتان وتفتقدان لأكثر الشروط أهمية: ما هي التجريدية المُقدمة عبر واجهة الإجرائية؟ إذا كانت فكرة التجريدية هي بأن تتوقع الإجرائية أن لديك ثلاث عناصر بيانات مُحددة، وليس من قبيل المصادفة تزويد هذه العناصر الثلاثة من قبل نفس الكائن، عندها يجب عليك أن تُمرر العناصر الثلاث المحددة كل على حدى. على كل حال، إذا كانت التجريدية هي بأنه سيكون لديك دائماً ذلك الكائن المعين، بمتناول اليد، وستقوم الإجرائية بشيء ما أو آخر مع ذلك الكائن، وبعدها، في الواقع، ستنكسر الفكرة التجريدية عندما تقوم بإظهار عناصر البيانات المحددة الثلاثة.

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول كيفية التفكير فير الواجهات، انظر "التجريد الجيد" في القسم 6.2.

إذا كنت تُمرر كامل الكائن ووجدت نفسك تُنشئ الكائن، واضعاً فيه العناصر الثلاثة المُحتاجة من قبل الإجرائية المُستدعاة، وبعدها تسحب هذه العناصر خارج الكائن بعد أن يتم استدعاء الإجرائية، فهذه إشارة بأنه عليك تمرير العناصر الثلاث المحددة بدلاً من كامل الكائن. (في العموم، إن وجود الشفرة التي "تُهيئ" من اجل استدعاء إجرائية أو "تنزع" بعد استدعاء الإجرائية هو إشارة بأن الإجرائية ليست مصممة بشكل جيد).

إذا وجدت نفسك تبدل بشكل متكرر قائمة الوسطاء للإجرائية، بوسطاء تأتي من نفس الكائن كل مرة، هذه إشارة على ان عليك تمرير كامل الكائن بدلاً من تمرير عناصر محددة.

استخدم الوسطاء المُسماة . في بعض لغات البرمجة، تستطيع بشكل واضح ضم الوسطاء الرسمية مع الوسطاء الفعلية. هذا يجعل استخدام الوسطاء ذاتي التوثيق بشكل أكبر، ويساعد على تجنب الأخطاء من الوسطاء غير المتطابقة. هنا مثال بلغة فيجول بيسك:

مثال بلغة فيجول بيسك لتحديد الوسطاء بوضوح

```
Private Function Distance3d( _
ByVal xDistance As Coordinate ،_
ByVal yDistance As Coordinate ،_
ByVal zDistance As Coordinate _
)
...
End Function
...
Private Function Velocity( _
ByVal latitude as Coordinate ،_
ByVal longitude as Coordinate ،_
ByVal elevation as Coordinate _
)
...
Distance = Distance3d( xDistance := latitude ،yDistance := longitude ،_
zDistance := elevation )
...
End Function
```

هنا مكان التصريح عن الوسطاء الرسمية.

هنا يتم ربط الوسطاء الفعلية إلى الوسطاء الرسمية.

هذه التقنية مفيدة بشكل خاص عندما يكون لديك قوائم أطول من المعدل للمعامِلات المتطابقة، والتي تزيد من فرص إدخالك لوسيط غير مطابق دون أن يكتشف ذلك المترجم البرمجي. بصراحة، يمكن أن يكون ربط الوسطاء مبالغاً فيه في العديد من البيئات البرمجية ولكن في حدود السلامة، أو في البيئات الأخرى عالية الموثوقية، فإن التأكيد الإضافي بأن الوسطاء تتطابق مع الطريقة التي تتوقعها يمكن أن تكون جديرة بالاهتمام.

تأكد من أن الوسطاء الفعلية تطابق الوسطاء الرسمية. الوسطاء الرسمية، المعروفة أيضاً كـ " الوسطاء الزائفة" هي المتغيرات المُصرح عنها في تعريف الإجرائية. الوسطاء الفعلية هي المتغيرات، أو الثوابت، أو التعابير المستخدمة في استدعاءات الإجرائية الفعلية.

خطأ شائع هو وضع نوع خاطئ من المتغيرات في استدعاء الإجرائية - مثلاً، استخدام عدد صحيح عند الحاجة لعدد حقيقي ذو الفاصلة العائمة float. (هذه المشكلة فقط في اللغات البرمجية ضعيفة الكتابة مثل لغة سي عندما لا تستخدم كامل تحذيرات المترجم. بينما لا تملك اللغات قوية مثل سي++ وجافا هذه المشكلة). عندما تكون المعاملات هي معاملات دخل فقط، هذا نادراً ما يكون مشكلة؛ عادةً يحول المترجم الأنواع الفعلية إلى أنواع رسمية قبل تمريرها للإجرائية. إذا كانت هذه مشكلة، عادةً يُعطيك المترجم تحذيراً. ولكن في بعض الحالات، بشكل مفرد عندما تكون المعامِلات مستخدمة لكلا الدخل والخرج، يمكنك أن تتأذى عن طريق تمرير النوع الخاطئ من المعاملات.

طَوّرْ عادةً فحص أنواع المعاملات في قوائم الوسطاء وانتبه إلى تحذيرات المترجم حول أنواع الوسطاء غير المتطابقة.

6.7 اعتبارات خاصة في استخدام التوابع

تدعم لغات البرمجة الحديثة مثل سي ++ وجافا وفيجول بيسك كلا التوابع والإجراءات. التابع هو الإجرائية التي تُعيد قيمة؛ الإجراء (procedure) هو الإجرائية التي لا تُعيد قيمة. بلغة سي++، تُسمى كل الإجرائيات نموذجياً "توابع"؛ على كل حال، نوع التابع الذي يُعيد قيمة خالية (void) يدل على إجراء. إن الاختلاف بين التوابع والإجراءات هو فرق أكثر دلالية كالفرق النحوي. وعلم الدلالات يجب أن يكون دليلك.

متى نستخدم التابع ومتى نستخدم الإجراء

يتجادل الأصوليين بأن التابع يجب أن يُعيد قيمة واحدة فقط، تماماً كما يفعل التابع الرياضي. هذا يعني بأنه يمكن للتابع أن يأخذ فقط وسطاء الدخل ويعيد قيمتها الوحيدة عبر التابع نفسه. يجب أن يُسمّى التابع دائماً حسب القيمة التي يعيدها، مثل sin() و CustomerID() و ScreenHeight(). يمكن للإجراء، من جهة أخرى، أخذ وسطاء دخل، ووسطاء تعديل، ووسطاء خرج – بالعدد الذي يريده من كل منها.

إن الممارسة البرمجية الشائعة هي بامتلاك تابع يعمل كإجراء ويعيد قيمة الحالة. منطقياً، يعمل هذا كإجراء، ولكونه يعيد قيمة، فهو بشكل رسمي تابع. على سبيل المثال، قد يكون لديك إجرائية تُدعى FormatOutput()، تُستخدم مع كائن تقرير بعبارات كهذه:

```
if ( report.FormatOutput( formattedReport ) = Success ) then…
```

في هذا المثال، تعمل report.FormatOutput() كإجراء، كونه يحوي وسطاء خرج، FormattedReport، ولكنه تقنياً تابع، لأن الإجرائية نفسها تعيد قيمة. هل هذه هي الطريقة الصحيحة لاستخدام تابع؟ في الدفاع عن هذه الطريقة، تستطيع الحفاظ على أن التابع يعيد قيمة ليس لها عمل مع الغاية الرئيسية للإجرائية، مثل تهيئة الخرج، أو مع اسم الإجرائية، report.FormatOutput(). في هذه الحالة، تعمل الإجرائية أكثر كما يعمل الإجراء حتى لو كانت تقنياً تابع. إن استخدام القيمة المُعادة للإشارة إلى نجاح أو فشل الإجراء ليس أمراً مزعجاً إذا تم استخدام التقنية بشكل متناسق.

البديل هو بإنشاء إجراء يحتوي على متغير حالة، كوسيط واضح، يعزز الشفرة مثل هذا الجزء:

```
report.FormatOutput( formattedReport, outputStatus )
if ( outputStatus = Success ) then…
```

أنا أُفضّل النموذج الثاني من التشفير، ليس لأنني عنيد تجاه الفرق بين التوابع والإجراءات، ولكن لأنها تجعل التفريق واضح بين استدعاء الإجرائية وفحص قيمة الحالة. يزيد الجمع بين الاستدعاء والفحص، بسطر واحد من النص البرمجي من كثافة العبارة، وبالمقابل إنه أمر معقد. والاستخدام التالي للتابع جيد أيضا:

```
outputStatus = report.FormatOutput( formattedReport )
if ( outputStatus = Success ) then…
```



بالمُختصر، استخدِم التابع إذا كانت الغاية الرئيسية للإجرائية هي إعادة القيمة المُشار إليها في اسم التابع. وإلا فاستخدم الإجراء.

ضبط القيمة المُعادة من التوابع

ينشئ استخدام تابع خطر إعادة التابع قيمة مُعادة خاطئة. وهذا يحدث عادةً عندما يكون للتابع العديد من المسارات المُحتملة وأحد هذه المسارات غير مُعد ليعيد قيمة. لتقليل هذا الخطر، قم بما يلي:

افحص كل مسارات الإعادة المُحتملة. عند إنشاء تابع، أنجز ذهنياً كل مسار للتأكد من أن التابع يُعيد قيمة في جميع الظروف الممكنة. إن تهيئة القيمة المعادة في بداية التابع لقيمة افتراضية عادة عملية جيدة –يوفر هذا شبكة أمان في حالة عدم تعيين قيمة الإعادة الصحيحة.

لا تُعيد مرجعيات أو مؤشرات للبيانات المحلية. حالما تنتهي الإجرائية وتخرج البيانات المحلية من المجال، سيصبح المرجع أو المؤشر للبيانات المحلية غير صالح. إذا احتاج كائن لإعادة معلومات حول بياناته الداخلية، يجب أن تُحفظ المعلومات كبيانات صف. يجب عليه بعدها أن يزود توابع وصول، تُعيد قيم عناصر بيانات العضو بدلاً من إعادة المرجعيات أو المؤشرات للبيانات المحلية.

7.7 اجرائيات الماكرو والإجرائيات المضمّنة[1]

تتطلب الإجرائيات المُنشأة بواسطة وحدات المسجل "الماكرو" (macros) للمعالج التمهيدي بعض الشروط الفريدة. تتعلق القواعد والأمثلة التالية باستخدام المعالج التمهيدي في لغة البرمجة سي++. إذا كنت تستخدم لغة برمجة أخرى أو معالج تمهيدي آخر، لائم القواعد مع الوضع الخاص بك.

ضع تعابير الماكرو بشكل كامل بين أقواس. لأن الماكرو ومعاملاتها تتمدد داخل الشفرة؛ فاحرص على أن تتمدد بالطريقة التي تريدها أنت. تكمن إحدى المشاكل الشائعة في إنشاء ماكرو كهذا:

مثال بلغة البرمجة سي++ عن ماكرو، لا يمتد بشكل صحيح

```
#define Cube( a ) a*a*a
```

إذا مررت لهذا الماكرو قيم غير رقمية لأجل a، فإنها لن تقوم بعملية الضرب بشكل صحيح. وإذا استخدمت التعبير Cube(x+1)، فإنها تتمدد إلى X+1*X+1*X+1، والذي، بسبب أسبقية عملية الضرب والجمع، ليس ما تريده. الأفضل، ولكنه لا يزال غير كامل، نسخة من الماكرو كهذه:

مثال بلغة البرمجة سي++ عن ماكرو لايزال لا يمتد بشكل صحيح

```
#define Cube( a ) (a)*(a)*(a)
```

هذا أفضل، ولكنه لا يزال غير كامل. إذا استخدمت Cube() في تعبير يحوي عمليات ذات أسبقية أعلى من أسبقية الضرب، ستنفصل الـ (a)*(a)*(a) عن بعضها. لمنع ذلك، طوّق كامل التعبير بأقواس:

مثال بلغة البرمجة سي++ عن ماكرو يعمل

```
#define Cube( a ) ((a)*(a)*(a))
```

أحط عبارات الماكرو-المتعددة بأقواس معقوفة. يمكن أن يحوي الماكرو عبارات متعددة، والتي ستؤدي إلى مشكلة إذا تعاملت معها كما لو أنها عبارة واحدة. فيما يلي مثال عن ماكرو يؤدي إلى متاعب:



مثال بلغة البرمجة سي++ عن ماكرو غير عامل مع عبارات متعددة

```
#define LookupEntry( key , index )\
index = (key - 10) / 5\;
index = min( index, MAX_INDEX )\;
```

[1] **إشارة مرجعية**: حتى إذا لم يكن لديك معالج تمهيدي للماكرو، يمكنك بناء واحد خاص بك. لمزيد من التفاصيل، انظر القسم 5.30 " بناء أدوات البرمجة الخاصة بك "

```
index = max( index, MIN_INDEX );
…
for ( entryCount = 0; entryCount < numEntries; entryCount++ )
LookupEntry( entryCount, tableIndex[ entryCount ] );
```

يتجه هذا الماكرو نحو المتاعب لأنه لا يعمل كتابع منتظم. كما هو مبين: الجزء الوحيد من الماكرو المنفذ في حلقة for هو السطر الأول من الماكرو:

```
index = (key - 10) / 5;
```

لتجنب هذه المشكلة، أحط الماكرو بأقواس معقوفة:

مثال بلغة البرمجة سي ++ عن ماكرو عامل مع عبارات متعددة

```
#define LookupEntry( key, index )\}
index = (key - 10) / 5\;
index = min( index, MAX_INDEX )\;
index = max( index, MIN_INDEX )\;
{
```

تُعتبر، بالعموم، العادة العملية باستخدام الماكرو كبديل عن استدعاء التابع عادة خطيرة وصعبة الفهم -عادة برمجية سيئة- لذا استخدم هذه التقنية إذا كانت الظروف الخاصة بك تتطلب ذلك.

سمّي مسجلات الماكرو التي تمتد للشفرة كالإجرائيات، وهكذا يمكن استبدالهم بواسطة الإجرائيات عند الضرورة. إن الاصطلاح بلغة البرمجة سي++ لتسمية الماكرو هي باستخدام كل الأحرف الكبيرة في الاسم. إذا كان بالإمكان استبدال الماكرو بواسطة إجرائية، على كل حال، سمّيه مُستخدما اصطلاح تسمية الإجرائية. بهذه الطريقة يمكنك استبدال الماكرو بالإجرائية والعكس بالعكس بدون تغيير أي شيء ماعدا الإجرائية المعنية.

ينطوي اتباع هذه النصائح على بعض المخاطر. إذا كنت عادةً تستخدم ++ و -- كتأثيرات جانبية (كجزء من عبارات أخرى)، فستتأذى من استخدام مسجلات الماكرو التي تفكر فيها على أنها إجرائيات. بالأخذ بعين الاعتبار المشاكل الأخرى مع الآثار الجانبية، وهذا هو سبب آخر لتجنب استخدام الآثار الجانبية.

القيود المفروضة على استخدام إجرائيات الماكرو

توفر اللغات الحديثة مثل ال سي++ الكثير من البدائل لاستخدام الماكرو:

- const للتصريح عن القيم الثابتة
- inline لتعريف التوابع التي سيتم ترجمتها كشفرة مضمنة

- template لتعريف العمليات القياسية مثل: max،min، وهكذا، بطريقة أمنة
- enum لتعريف الأنواع التعدادية
- typedef لتعريف بدائل نوع بسيط



كما قال بيارن ستروستروب، مُصمم لغة البرمجة سي++، " تقريبا يُظهر كل ماكرو خلل في لغة البرمجة، أو في البرنامج، أو في المبرمج... عندما تقوم باستخدام الماكرو، يجب أن تتوقع خدمة رديئة من الأدوات، مثل المصححات، والأدوات المرجعية، والراسمات" (ستروستروب 1997). إن الماكرو مفيد لدعم التصنيف المشروط – انظر القسم 8.6، "مساعدات التصويب" – ولكن يستخدم عادة المبرمج الحَذِر الماكرو كبديل للإجرائية فقط كحل أخير.

الإجرائيات على السطر Inline Routines

تدعم لغة البرمجة سي++ الكلمة المفتاحية `inline`. تسمح الإجرائيات على السطر للمبرمج بالتعامل مع الشفرة كإجرائية في وقت كتابة الشفرة، ولكن عادةً يحول المترجم البرمجي كل نسخة من الإجرائية إلى شفرة على السطر في وقت الترجمة. تقول النظرية أنه يستطيع `inline` المساعدة بإنتاج شفرة عالية الكفاءة، تتجنب عبء استدعاء الإجرائية.

استخدم الإجرائيات على السطر بشكل معتدل. تنتهك الإجرائيات المضمنة مبدأ التغليف، لأن لغة البرمجة سي++ تتطلب من المبرمجين وضع النص البرمجي المتعلق بتنفيذ الإجرائية المضمنة داخل الملف الرأسي، الأمر الذي يكشف النص البرمجي لكل مبرمج يستخدم الملف الرأسي.

تتطلب الإجرائيات المُضمنة توليد كامل شفرة الإجرائية في كل مرة يتم فيها استدعاء الإجرائية، والذي سيزيد من حجم الشفرة من أجل إجرائية مُضمنة من أي حجم. وهذا يمكن أن يخلق المشاكل الخاصة بها.

إن السطر الأخير في التضمين، ولأسباب الاداء، هو نفس السطر الأخير لأي تقنية كتابة شفرة أخرى محفزة بواسطة الاداء[1]: "ارسم" الشفرة وقس التحسن. إن لم تبرر مكاسب الاداء المنتظرة الإزعاج من "رسم" الشفرة للتأكد من التحسن، فإنها لا تبرر التآكل جودة الشفرة أيضاً.

لائحة اختبار: الإجرائيات عالية الجودة[2]
قضايا الصورة الكبيرة

[1] المقصود بالرسم هنا: اخضاع الشفرة لراسم، لتحليل أداء الشفرة

`cc2e.com/07922`

إشارة مرجعية: هذه عبارة عن لائحة اختبار للاعتبارات حول جودة الإجرائية. من أجل قائمة الخطوات المستخدمة في بناء إجرائية، انظر قائمة التحقق "عملية برمجة الشفرة الزائفة" في الفصل 9

- هل سبب إنشاء الاجرائية كافٍ؟
- هل تم وضع جميع أجزاء الاجرائية، التي ستستفيد من وضعها في الإجرائيات الخاصة بها، في الإجرائيات الخاصة بها؟
- هل اسم الإجرائية اسم قوي واضح، وهل هو عبارة عن (فعل + اسم كائن) لإجراء أو هو وصف للقيمة التي يعيدها تابع؟
- هل يصف اسم الإجرائية كل شيء تقوم به الإجرائية؟
- هل أسست اصطلاحات لتسمية العمليات الشائعة؟
- هل لدى الاجرائية قوة التماسك الوظيفي — تنفذ شيء واحد وشيء واحد فقط مع إنجازه بشكل جيد؟
- هل لدى الاجرائية اقتران ضعيف— هل اتصالات الإجرائية مع الإجرائيات الأخرى قليل وحميم ومرئي ومرن؟
- هل طول الإجرائية يُحدّد طبيعيا بوظيفته ومنطقه، أكثر من تحديده عبر المعايير الصناعية لكتابة الشفرة؟

قضايا تمرير الوسيط

- هل قائمة وسطاء الإجرائية، تُؤخذ ككل متكامل، وتُقدم واجهة تجريدية متناسقة؟
- هل قائمة وسطاء الإجرائية مرتبة بشكل معقول، وتتضمن مطابقة الترتيب للوسطاء في الإجرائيات المتشابهة؟
- هل تم توثيق افتراضيات الواجهة؟
- هل للإجرائية سبعة وسطاء أو أقل؟
- هل كل وسيط دخل مُستخدم؟
- هل كل وسيط خرج مُستخدم؟
- هل تتجنب الإجرائية استخدام وسطاء الدخل كمتغيرات عاملة؟
- إذا كانت الإجرائية تابع، هل تُعيد قيمة صالحة في جميع الظروف الممكنة؟

نقاط مفتاحية

- السبب الأكثر أهمية لإنشاء إجرائية هو تحسين الإدارة الفكرية للبرنامج، كما يمكنك إنشاء إجرائية للكثير من الأسباب الجيدة الأخرى. توفير المساحة هو سبب ثانوي؛ تحسين إمكانية القراءة والفاعلية وإمكانية التعديل هي أسباب أفضل.
- أحياناً العملية، التي أغلب فوائدها من كونها موضوعة في إجرائية خاصة بها، هي عملية بسيطة.

- يمكنك تصنيف الإجرائيات إلى أنواع متعددة من ناحية التماسك، ولكنه يمكنك جعل أغلب الإجرائيات متماسكة وظيفياً، وهو الأفضل.
- يدُل اسم الإجرائية على جودتها. إذا كان الاسم سيء ودقيق، قد تكون الإجرائية مُصممة بشكل سيء. وإذا كان الاسم سيء وغير دقيق، فإنه لا يخبرك بما يفعله البرنامج. بطريقة أخرى، يعني الاسم السيئ بأنه يجب تغيير البرنامج.
- يجب اســتخدام التوابع فقط عندما يكون الهدف الأســاســي من التابع هو إعادة القيمة المحددة والموصوفة باسم التابع.
- يستخدم المبرمجون الحريصون الإجرائيات ماكرو بحذر وفقط كحل أخير.

الفصل الثامن

البرمجة الوقائيّة

المحتويات[1]

مواضيع ذات صلة

- إخفاء المعلومات: " إخفاء الأسرار (إخفاء المعلومات)" في المقطع 3.5
- التصميم من أجل التغيير: " تحديد المناطق المحتمل تغييرها" في المقطع 3.5
- هيكلة البرمجية: الجزء 3.5
- التصميم في البناء: الفصل 5
- التنقيح: الفصل 23

نقطة مفتاحية

البرمجة الوقائية لا تعني بأن تكون مدافعا عن برمجتك – " هذا لا يعمل!" تقوم الفكرة على القيادة الوقائية. في القيادة الوقائية، أنت تتخذ عقليّة أنّك لست متأكداً مما سيفعله السائقين الآخرين. بتلك الطريقة، ستتأكد بأنهم إذا قاموا بفعل شيء ما خطير فإنك لن تتأذى. فأنت تتحمل مسؤولية حماية نفسك حتى عندما يكون ذلك الخطأ ناتج عن سائق آخر. في البرمجة الوقائية، الفكرة الرئيسيّة هي أنّه إذا مرّرت الإجرائيات

[1] cc2e.com/0861

بيانات سيئة، فإن ذلك لن يكون مؤذياً، حتى لو كانت البيانات السيّئة هي خطأ إجرائيات أخرى. وبشكل أكثر عموميّة، إنّها الإدراك بأن البرامج ستواجه مشكلات وتعديلات، وأنّ المبرمج الجيد سيطوّر الشفرة وفقاً لذلك.

يصف هذا الفصل كيفيّة حماية نفسك من الضعف، وعالَم البيانات غير الصالحة القاسي، والأحداث التي لا يمكن أن تحدث، وأخطاء المبرمجين الآخرين. إذا كنت مبرمجاً متمرساً، ربما ستتخطى القسم التالي حول التعامل مع بيانات الإدخال وتبدأ بالجزء 8.2، الذي يستعرض استخدام التأكيدات.

8.1 حماية برنامجك من الإدخالات غير الصالحة[1]

ربما سمعت هذا التعبير في المدرسة، " دخل سيء يعطي خرج سيء" Garbage in, garbage outهذا التعبير هو أساسا نسخة تطوير البرمجيات على مسؤولية الزبون. دع المستخدم حذراً.



دخل سيء، خرج سيء ليست جيدة بما فيه الكفاية لإنتاج البرمجيات. البرنامج الجيد ينتج خرجاً جيداً دئماً، بغض النظر عمّا الذي يتطلبه ذلك. البرنامج الجيد يستخدم " دخل سيء، لا شيء في الخارج،" أو " دخل سيء، تخرج رسالة خطأ،" أو عوضاً عن ذلك " لا يسمح للدخل السيء". وفقاً لمعايير اليوم، " دخل سيء يعطي خرج سيء " هو علامة سيئة، وبرنامج غير آمن.

هناك ثلاث طرق عامة للتعامل مع المدخلات السيئة:

تحقق من قيم كل البيانات القادمة من مصادر خارجية عند الحصول على البيانات من ملف، أو مستخدم، أو الشبكة، أو من واجهة خارجية أخرى، افحصها لتتأكد من أن البيانات تقع ضمن المجال المسموح به. تأكد من أنّ القيم العددية ضمن مجال التسامح وأن هذه السلاسل المحرفية قصيرة بما فيه الكفاية للتعامل معها. إذا أرادت السلسلة المحرفية أن توضح نطاقاً مقيداً من القيم (مثل معرّف معاملة مالية أو شيء مشابه)، تأكّد من أن تلك السلسلة صالحة للغرض المطلوب منها؛ وإلا قم برفضها. إذا كنت تعمل على تطبيق آمن، كن حذراً بشكل خاص من البيانات التي ربما قد تهاجم النظام الخاص بك:

كمحاولة إغراق التخزين المؤقت، وحقن أوامر SQL، وحقن HTML أو شفرة XML، وتجاوز العدد الصحيح، والبيانات الممرّرة إلى استدعاءات النظام، وهلم جرّا.

تحقق من قيم كل وسطاء إدخال الإجرائيات التحقق من قيم وسطاء إدخال الإجرائيات هو أساساً نفس فحص البيانات التي تأتي من مصدر خارجي، باستثناء أنّ البيانات تأتي من إجرائيات أخرى بدلاً من الواجهة الخارجية. تؤمن المناقشة في الفصل 8.5، " حصّن برنامجك لاحتواء الأضرار الناتجة عن الأخطاء،" طريقة عمليّة لتحديد أي الإجرائيات التي ستحتاجها لفحص مدخلاتها.

[1] عندما تكون البيانات المدخلة غير كاملة وغير دقيقة، سيكون الخرج الناتج غير موثوق وبلا قيمة

قرر كيف ستتعامل مع المدخلات السيئة عندما تكتشف وسيط غير صالح، ما الذي ستفعله مع ذلك؟ اعتماداً على الحالة، يمكن أن تختار أي من مجموعة النُهج المختلفة، التي تم وصفها بالتفصيل لاحقاً في هذا الفصل في الجزء 3.8 " تقنيات معالجة الأخطاء".

البرمجة الوقائية فعّالة كمساعد على تقنيات تحسين الجودة الأخرى الموضّحة في هذا الكتاب. إنّ أفضل شكل لكتابة الشفرة الوقائية هو عدم إدراج الأخطاء في المقام الأول. إنّ استخدام التصميم التكراري، وكتابة الشفرة الزائفة "Pseudocode" قبل كتابة الشفرة، وكتابة حالات الاختبار قبل كتابة الشفرة، وإجراء فحص تصميم منخفض المستوى جميعها نشاطات تساعد على منع إدراج العيوب. لذلك ينبغي أن يُمنَحوا أولوية أعلى من البرمجة الوقائية. لحسن الحظ، تستطيع استخدام البرمجة الوقائية مدموجة مع التقنيات الأخرى.

كما يوحي الشكل 8-1، حماية نفسك من المشاكل التي تبدو صغيرة يمكن أن تحدث فرقا أكبر مما تعتقد. يصف باقي هذا الفصل خيارات محدّدة لفحص البيانات من مصادر خارجية، وفحص وسطاء الإدخال، والتعامل مع المدخلات السيئة.



الشكل 8-1 غرق جزء من الجسر العائم على الطريق-90 في سياتل خلال عاصفة لأن خزانات الطفو تُركت مكشوفة، وامتلأت بالمياه، وأصبح الجسر ثقيلاً جداً على الطفو. خلال البناء احمي نفسك من الأشياء الصغيرة التي قد تُهِمّ أكثر مما تتصور.

2.8 التأكيدات (المصادقات)

التأكيد أو هو شفرة يتم استخدامها خلال التطوير – عادةً إجرائية أو مُسجل "ماكرو "macro – والذي يسمح للبرنامج أن يفحص نفسه عندما يتم تشغيله. عندما يكون التأكيد صحيحاً، فهذا يعني أنّ كل شيء يعمل كما هو متوقع. بينما عندما يكون خاطئاً، فهذا يعني بأنّه اكتُشف خطأ غير متوقع في الشفرة. على سبيل المثال، إذا كان النظام يفترض أن ملف معلومات الزبون لن يحتوي على أكثر من 50000 سجل، يمكن أن يحتوي البرنامج على تأكيد أن عدد السجلات هو أقل أو مساوٍ لـ 50000. التأكيد سيبقى صامتاً طالما أن عدد السجلات أقل أو يساوي 50000. على أية حال، إذا صادف أكثر من 50000 سجل، فأنّه "يؤكّد" بصوت عالٍ أن هناك خطأ في البرنامج.



إنّ التأكيدات مفيدة بشكل خاص في البرامج الكبيرة والمعقّدة وفي البرامج ذات الموثوقية العالية. فهي تمكّن المبرمجين من سرعة استبعاد فرضيات الواجهة غير المطابقة، والأخطاء التي قد تحدث عندما يتم تعديل الشفرة، وما إلى ذلك.

يأخذ التأكيد عادة اثنين من المُعاملات: التعبير المنطقي الذي يصف الافتراض الذي يُفترض أن يكون صحيحاً، ورسالة للعرض إذا لم يكن ذلك صحيحاً. وإليك كيف سيبدو تأكيد جافا إذا كان من المتوقع أن يكون القاسم المتغير غير صفري:

مثال جافا على استخدام التأكيد

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

يؤكد هذا التأكيد أن القاسم لا يساوي 0. المعامل الأول، القاسم !=0، هو تعبير منطقي يقيّم كـ صحيح أو خاطئ. المعامل الثاني هو رسالة للطباعة في حال كان الجدال الأول خاطئ– هذا هو، إذا كان التأكيد خاطئ.

استخدام التأكيدات لتوثيق الافتراضات التي تم اجرائها في الشفرة واستبعاد الحالات غير المتوقعة. التأكيدات يمكن أن تستخدم للتحقق من افتراضات من هذا القبيل.

- أنّ قيمة وسيط الدخل تقع ضمن مجالها المتوقع (أو أن قيمة وسيط الخرج كذلك).
- أن الملف أو التدفق يكون مفتوح (أو مغلق) عندما يبدأ الإجرائية التنفيذ (أو عندما ينهي التنفيذ).
- أنّ الملف أو التدفق يكون في البداية (أو النهاية) عندما يبدأ الإجرائية التنفيذ (أو عندما ينهي التنفيذ).
- أنّ الملف أو التدفق مفتوح للقراءة فقط، أو للكتابة فقط، أو لكلٍ من القراءة والكتابة.
- أنّ القيمة لمتحول الدخل-فقط لا تتغير بواسطة الإجراءات التكرارية.
- أنّ المؤشر غير فارغ.

- أنّ مصــفوفة أو حاوية container أخرى تم تمريرها إلى إجرائية يمكن أن تحتوي على الأقل "س" عدد من عناصر البيانات.
- أنّ الجدول قد تم تهيئته ليحتوي قيم حقيقية.
- أنّ الحاوية فارغة (أو ممتلئة) عندما تبدأ الإجرائية التنفيذ (أو عندما تنتهي).
- أنّ النتائج من الإجرائيات المعقّدة فائقة التحسين تطابق النتائج من إجرائيات أبطأ لكن مكتوبة بشكل واضح.

بالطبع، هذه مجرد أساسيات، وستحتوي إجرائياتك الخاصّة العديد من الافتراضات الأكثر تحديداً والتي تستطيع توثيقها باستخدام التأكيدات.

عادةً، أنت لا تريد أن يرى المستخدمون رسائل التأكيد في الشفرة النهائية؛ التأكيدات هي بشكل رئيسي للاستخدام خلال التطوير والصيانة. وعادة ما يتم ترجمة التأكيدات في الشفرة خلال وقت التطوير ويتم ترجمتها خارج الشفرة من أجل الإنتاج. خلال التطوير، تزيل التأكيدات الافتراضات المتناقضة، والحالات غير المتوقعة، والقيم السيئة الممرّرة إلى الإجرائيات، إلخ. وخلال الإنتاج، يمكن ترجمتها خارج الشفرة بحيث لا تخفّض التأكيدات من أداء النظام.

آلية بناء التأكيد الخاص بك[1]

تملك العديد من اللغات دعما للتأكيدات مضمّنا داخلها، بما في ذلك سي++، وجافا، ومايكروسوفت فيجوال بيسك. إذا لم تدعم لغتك تأكيدات الإجرائيات بشكل مباشر، فإنّها سهلة الكتابة. معيار سي++ يؤكد أن الماكرو غير مؤمّن للرسائل النصيّة. هنا مثال عن **تأكيد** مطوّر ينفّذ كماكرو سي++:

مثال سي ++ لماكرو تأكيد

```
C++ Example of an Assertion Macro
#define ASSERT( condition, message ) {        \
   if ( !(condition) ) {                      \
       LogError( "Assertion failed: ",        \
           #condition, message );             \
       exit( EXIT_FAILURE );                  \
    }                                         \
}
```

[1] **إحالة مرجعية** بناء تأكيد الإجرائيات الخاصة بك هو مثال جيد على البرمجة "إلى" اللغة أكثر منها فقط البرمجة "في" اللغة. لمزيد من التفاصيل عن الفارق، انظر الجزء 34.4."برمج إلى لغتك، وليس فيها"

إرشادات لاستخدام التأكيدات

ها هي بعض الإرشادات لاستخدام التأكيدات:

استخدم شفرة معالجة الأخطاء من أجل الحالات التي تتوقع حدوثها؛ استخدم التأكيدات من أجل الحالات التي لا يجب أن تحدث، التأكيدات تتحقق من الحالات التي يجب ألا تحدث. شفرة التعامل مع الأخطاء تتحقق من الظروف غير الاسمية `off-nominal` التي لا تحدث غالباً، لكنها

متوقعة من قبل المبرمج الذي كتب الشفرة والتي تحتاج لأن يتم التعامل معها من قبل شفرة الانتاج. تتحقق معالجة الأخطاء عادة من بيانات الادخال السيئة؛ بينما تتحقق التأكيدات من الثغرات في الشفرة.

إذا تم استخدام شفرة معالجة الأخطاء لمعالجة حالة شاذة، سيمكّن التعامل مع أخطاء البرنامج من الاستجابة إلى الخطأ بمرونة. لكن إذا أطلق التأكيد من أجل حالة شاذة، فأن عملية التصحيح لا تكون بمجرد التعامل مع الخطأ بمرونة – بل تكون بتغيير الشفرة المصدرية للبرنامج، وإعادة الترجمة، وإطلاق نسخة جديدة من البرمجية.

يمكن التفكير بالتأكيدات بطريقة أخرى جيدة وهي كتوثيق قابل للتنفيذ- لا يمكنك الاعتماد عليهما لتجعل الشفرة تعمل، لكن تستطيع توثيق الافتراضات بشكل أكثر فاعليّة مما تستطيعه تعليقات لغة البرمجة.

تجنّب وضع شفرة قابلة للتنفيذ في التأكيدات وضع الشفرة في التأكيد ترفع من إمكانية أن يزيل المترجم الشفرة عندما توقف التأكيد. افرض أن لديك تأكيد مثل هذا:

مثال فيجوال بيسك على استخدام خطير لتأكيد[1]

```
Debug.Assert( PerformAction() ) ' Couldn't perform
action
```

المشكلة في هذه الشفرة هي أنّه، إذا لم تقم بترجمة التأكيدات، فأنت لا تترجم الشفرة التي تنجز العمل. ضع العبارات القابلة للتنفيذ على سطورها الخاصة، وأسند النتائج إلى الحالات المتغيرة، واختبر الحالات المتغيرة عوضاً عن ذلك. هنا مثال على الاستخدام الآمن للتأكيد:

مثال فيجوال بيسك لاستخدام آمن لتأكيد

```
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

[1] **إشارة مرجعية** تستطيع أن تظهر هذا كواحدة من العديد من المشاكل المرتبطة بوضع عبارات متعددة في سطر واحد. لأمثلة أكثر انظر " استخدم عبارة واحدة فقط في السطر" في الجزء 5.31

استخدم التأكيدات للتوثيق ولإثبات الشروط المسبقة والشروط اللاحقة[1] الشروط المسبقة والشروط اللاحقة هم جزء من النهج إلى تصميم وتطوير البرنامج يعرف كـ"التصميم بالاتفاق" (ميير 1997). عندما تستخدم الشروط المسبقة والشروط اللاحقة، كل إجرائية أو صف يصيغ الاتفاق مع بقية البرنامج.

الشروط المسبقة هي الخصائص التي تتعهد بأن تكون شفرة الزبون من الإجرائية أو الصف صحيحة قبل استدعاء الإجرائية أو تمثيل الكائن. الشروط المسبقة هي التزامات شفرة الزبون إلى الشفرة التي قامت باستدعائها.

الشروط اللاحقة هي الخصائص التي يتعهد الإجرائية أو الصف بأن تكون صحيحة عندما تُنهي التنفيذ. الشروط اللاحقة هي التزامات الصف أو الإجرائية إلى الشفرة التي تستخدمهم.

التأكيدات أداة مفيدة لتوثيق الشروط المسبقة والشروط اللاحقة. التعليقات يمكن أن تستخدم لتوثيق الشروط المسبقة والشروط اللاحقة، لكن، بخلاف التعليقات، التأكيدات يمكن أن تتحقق فيما إذا كانت الشروط المسبقة والشروط اللاحقة صحيحة بشكل ديناميكي.

في المثال التالي، استخدمت التأكيدات لتوثيق الشروط المسبقة والشروط اللاحقة لإجرائية *السرعة*:

مثال فيجوال بيسك لاستخدام التأكيدات لتوثيق الشروط المسبقة واللاحقة

```
Private Function Velocity ( _
   ByVal latitude As Single, _
   ByVal longitude As Single, _
   ByVal elevation As Single _
   ) As Single

   ' Preconditions
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )
   ...
   ' Postconditions Debug.Assert ( 0 <= returnVelocity And returnVelocity <=
600 )
   ' return value
   Velocity = returnVelocity
End Function
```

إذا كان متغيرات خط الطول، وخط العرض، والارتفاع تأتي من مصدر خارجي، يجب أن تفحص القيم غير الصالحة ويتم التعامل معها من قبل شفرة معالجة الأخطاء أكثر من التأكيدات. أما إذا جاءت المتغيرات من

[1] **اقرأ أيضا** للمزيد حول الشروط المسبقة، انظر بناء البرمجيات غرضية التوجه (مبيير 1997 Meyer)

مصدر داخلي موثوق. فإنّ تصميم الإجرائية يعتمد على الافتراض أنّ هذه القيم ستكون ضمن مجالاتها الصحيحة، عندها تكون التأكيدات مناسبة.

من أجل شفرة شديدة المتانة، تأكد ومن ثم عالج الخطأ بأية حال[1] من أجل أي حالة معطاة، الإجرائيات بشكل عام تستخدم إما التأكيد أو شفرة معالجة الأخطاء، لكن لا تستخدم كلاهما. ويرى بعض الخبراء أن هناك حاجة إلى نوع واحد فقط (مبير 1997).

لكن تميل البرامج والمشاريع في العالم الحقيقي لأن تكون أكثر فوضوية بالاعتماد على التأكيدات فقط. في نظام كبير طويل الأمد، قد يتم تصميم أجزاء مختلفة من قبل مصممين مختلفين على مدى فترة من 5-10 سنوات أو أكثر. سيكون المصممين منفصلين زمنياً، عبر العديد من الإصدارات. وسوف تركز تصميماتهم على التقنيات المختلفة لنقاط مختلفة في تطوير النظام.

سيكون المصممون منفصلين جغرافياً، خصوصاً إذا تم الحصول على أجزاء النظام من مصادر خارجية. وسيعمل المبرمجون على معايير تشفير مختلفة لنقاط مختلفة في عمر النظام.

في فريق التطوير الكبير، وسيكون حتماً بعض المبرمجين أكثر ضميراً من البعض الآخر وبعض أجزاء الشفرة سوف تُراجع بصرامة أكثر من الأجزاء الأخرى. بعض المبرمجين سيقومون باختبار الوحدة لشفرتهم بكُلّيّة أكثر من الآخرين. لا يمكنك الاعتماد على الشمولية، مع فِرق اختبار تعمل في مناطق جغرافية مختلفة وتخضع إلى ضغوط العمل التي تنتج عن مدى تغطية الاختبار التي تتغير مع كل إصدار. أو فحص انحدار مستوى النظام، وغيره.

في مثل هذه الظروف، كلاً من التأكيدات وشفرة معالجة الأخطاء يمكن استخدامها لمعالجة نفس الخطأ.[2] على سبيل المثال، في الشفرة المصدريّة لمايكروسوفت ووردWord، تم تأكيد الشروط التي يجب أن تكون دائما صحيحة، لكن مثل هذه الأخطاء تعالج من خلال شفرة معالجة الأخطاء في حال فشل التأكيد. تكون التأكيدات قيّمة بالنسبة للتطبيقات الكبيرة جداً والمعقدة وطويلة الأمد مثل وورد. لأنها تساعد على إزالة أكبر قدر ممكن من أخطاء زمن التطوير. لكن التطبيق معقد جداً (ملايين من أسطر الشفرة) وقد مرّت من خلال أجيال عديدة من التعديل فأنّه من غير الواقعي الاقتراض أن كل خطأ ممكن سيتم الكشف عنه وتصحيحه قبل إصدار البرمجية، وكثير من الأخطاء يجب أن تتم معالجتها في إصدار إنتاج النظام كذلك.

هنا مثال عن كيف يمكن العمل في مثال "velocity" السرعة:

[1] **إحالة مرجعية** لمزيد من المتانة انظر" المتانة ضد التصحيح" في الجزء 3.8، لاحقاً في هذا الفصل

[2] **اختبار الوحدة** Unit test: هو اختبار أن الشفرة تؤدي الغرض الذي صممت لأجله.

مثال فيجوال بيسك لاستخدام التأكيدات لتوثيق الشروط المسبقة واللاحقة

```
Private Function Velocity ( _
   ByRef latitude As Single, _
   ByRef longitude As Single, _
   ByRef elevation As Single _
   ) As Single
   ' شروط مسبقة
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )   |
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )
   ...
  'تعقيم بيانات الإدخال. ينبغي أن تكون القيم ضمن مجال التأكيد في الأعلى
   ' ولكن إذا لم تكن القيم ضمن المجال المقبول, سيتم تغييرها إلى أقرب قيمة
   'مقبولة
   If ( latitude < -90 ) Then
      latitude = -90
   ElseIf ( latitude > 90 ) Then
      latitude = 90
   End If
   If ( longitude < 0 ) Then
      longitude = 0
   ElseIf ( longitude > 360 ) Then
   ...
```

هنا شفرة التأكيد

هنا الشفرة التي تعالج المدخلات السيئة وقت التشغيل

3.8 تقنيات معالجة الأخطاء

تستخدم التأكيدات لمعالجة الأخطاء التي يجب ألا تحدث في الشفرة. كيف تعالج الأخطاء التي تتوقع حدوثها؟ بالاعتماد على ظروف محدّدة، ربما قد تريد إرجاع قيمة محايدة، أو الاستبدال بالقسم التالي من البيانات الصالحة، أو إرجاع نفس الإجابة كما في آخر مرّة سابقة، أو استبدال بأقرب قيمة مقبولة، أو تسجيل رسالة تحذير إلى ملف، أو إرجاع شفرة الخطأ، أو استدعاء إجرائية لمعالجة الخطأ أو الكائن، أو إرجاع رسالة الخطأ، أو الإغلاق – أو ربما قد ترغب باستخدام مزيج من هذه الاستجابات.

هنا تفاصيل أكثر على هذه الخيارات:

إرجاع قيمة محايدة في بعض الأحيان الاستجابة الأفضل للبيانات السيئة ببساطة هي مواصلة التشغيل وإرجاع القيمة المعروف بأنها غير مؤذية. قد يعيد الحساب العددي القيمة 0. قد ترجع العملية على السلسلة المحرفية سلسلة فارغة، أو قد يرجع تشغيل المؤشر مؤشر فارغ. قد يستخدم إجراء الرسم التكراري الذي يحصل على قيمة إدخال سيئة للون في لعبة فيديو لون الخلفية الافتراضي أو لون المقدمة. على كل حال،

في إجراء الرسم التكراري الذي يعرض بيانات الأشعة السينية x-ray لمرضى السرطان، لا تريد منه ابداً عرض قيمة "محايدة". في تلك الحالة، سيكون إيقاف تشغيل البرنامج أفضل من عرض بيانات غير صحيحة للمريض.

استبدال بالقسم التالي من البيانات الصالحة عند معالجة تدفق البيانات، تستدعي بعض الظروف ببساطة إرجاع البيانات الصالحة التالية. إذا كنت تقرأ السجلات من قاعدة البيانات وتواجه سجلاً معطوباً، فيمكنك ببساطة مواصلة القراءة حتى تجد سجلاً صالحاً. فإذا كنت تأخذ قراءات ميزان حرارة 100 مرة في الثانية ولم تحصل على قراءة صالحة لمرة واحدة، يمكنك ببساطة انتظار 100/1 جزء من الثانية آخر وآخذ القراءة التالية.

إرجاع نفس الإجابة كما في المرة السابقة إذا لم تستطع برمجية قراءة درجات الحرارة الحصول على قراءة في مرّة واحدة، من الممكن ببساطة الرجوع إلى آخر قيمة للقراءة السابقة. فاعتماداً على التطبيق، قد لا يكون من المرجح كثيراً أن تتغير درجة الحرارة في 100/1 من الثانية.

في لعبة الفيديو، إذا اكتشفت لون غير صالح في طلب لرسم جزء من الشاشة، يمكنك ببساطة استخدام نفس اللون المُستَخدم سابقاً.

ولكن إذا كنت تقوم بصفقة مالية في صرّاف الأوراق النقدية الآلي، فربما لن ترغب باستخدام "نفس الجواب كما في آخر مرة"-فهذا من شأنّه أن يكون رقم الحساب المصرفي للمستخدم السابق!

استبدال بأقرب قيمة مقبولة في بعض الحالات، قد تختار إرجاع أقرب قيمة مقبولة، كما هو الحال في مثال السرعة السابق. وهذا ما يكون غالباً نهجاً معقولاً عند أخذ قراءات من أداة معايرة. حيث يمكن معايرة ميزان الحرارة بين 0 و100 درجة مئوية، على سبيل المثال.

إذا اكشفت قراءة أقل من 0، يمكنك استبدالها بـ 0، وهي أقرب قيمة مقبولة. أما إذا اكشفت قيمة أكبر من 100، يمكنك استبدالها بـ 100. وبالنسبة لعملية السلسلة المحرفية، إذا تم الإبلاغ عن طول سلسلة أقل من 0، يمكنك استبدالها بـ 0.

سيارتي تستخدم هذا النهج في التعامل مع الخطأ كلما قُدتُ السيارة إلى الخلف. خلال ذلك لا يُظهر عداد السرعة الخاص بي سرعات سلبية، عندما أقود إلى الخلف فأنّه يُظهر ببساطة سرعة 0-أقرب قيمة مقبولة.

تسجيل رسالة تحذير إلى ملف عندما يتم الكشف عن بيانات سيئة، يمكنك اختيار تسجيل رسالة تحذير إلى ملف ثم المتابعة.

ويمكن استخدام هذا النهج بالاقتران مع تقنيات أخرى مثل الاستبدال بأقرب قيمة نظامية أو استبدالها بالجزء التالي من البيانات الصالحة. إذا كنت تستخدم سجل، انظر فيما إذا كان بإمكانك جعله متاحاً للعموم بأمان أو فيما إذا كنت بحاجة إلى تشفيره أو حمايته بطريقةٍ أخرى.

إرجاع رسالة خطأ يمكنك أن تقرر أن أجزاء معينة من النظام ستعالج الأخطاء. والأجزاء الأخرى لن تعالج الخطأ محليا، ستقوم ببساطة بالإبلاغ عن وجود خطأ مُكتشف وتثق بان بعض إجرائيات الأخرى ذات المستوى الأعلى في هرمية الاستدعاء ستعالج الخطأ. يمكن أن تكون الآلية المحددة لإشعار بقية النظام بحدوث خطأ أيًا مما يلي:

- تعيين قيمة متغير الحالة
- إرجاع حالة كقيمة معادة للدالة
- رمي استثناء باستخدام آلية الاستثناء المضمّنة للغة

وفي هذه الحالة، تكون آلية الإبلاغ عن الأخطاء المحدّدة أقل أهميّة من القرار المتعلق بأجزاء النظام التي ستتعامل مع الأخطاء مباشرة وتلك التي ستبلغ عن وقوعها فقط. إذا كان الأمان مهماً، تأكّد من أنّ إجرائيات الاتصال تحقق دائما شفرات الإرجاع.

استدعاء كائن/إجرائية معالجة الخطأ وهو نهج أخر لتركيز معالجة الخطأ في إجرائية عامة لمعالجة الأخطاء أو كائن لمعالجة الأخطاء. الإيجابية لهذا النهج تتلخص في أن المسؤولية عن معالجة الأخطاء يمكن أن تكون مركزية، الأمر الذي يجعل تصحيح الأخطاء أسهل.

والفكرة هنا أن كامل البرنامج سيعرف عن هذه الإمكانية المركزية وسوف يقترن بها.

إذا كنت تريد إعادة استخدام أيّاً من الشفرة البرمجية من النظام في نظام آخر، سيتوجب عليك سحب آلية معالجة الخطأ جنباً إلى جنب مع الشفرة التي تعيد استخدامها.

ينطوي هذا النهج على توريط أمني هام. إذا واجهت الشفرة تجاوز ذاكرة التخزين المؤقت، فمن المحتمل أن المهاجم قد اخترق عنوان الإجرائية أو الكائن المعالج.

وهكذا، بمجرد تجاوز ذاكرة التخزين المؤقت مرة خلال تشغيل التطبيق، فسيكون من غير الآمن استخدام هذا النهج.

عرض رسالة خطأ أينما تواجه الخطأ هذا النهج يقلّل من معالجة الأخطاء العامة؛ ومع ذلك، فأنّ لديه القدرة على نشر رسائل واجهة المستخدم خلال التطبيق بأكمله، والتي يمكن أن تخلق تحدّيات عندما تحتاج إلى إنشاء واجهة مستخدم متسقة، عند محاولة فصل واجهة المستخدم بشكل واضح عن بقية النظام، أو عند المحاولة لتوطين البرنامج إلى لغة مختلفة. أيضا، احذر من إخبار مهاجم محتمل للنظام أكثر من اللازم. حيث يستخدم المهاجمون أحياناً رسائل الخطأ لاكتشاف كيفيّة مهاجمة النظام.

التعامل مع الخطأ بالطريقة التي تعمل على نحو أفضل محلياً بعض التصاميم تدعو إلى التعامل مع جميع الأخطاء محلياً- القرار في اختيار منهج وسيط لاستخدامه لمعالجة الأخطاء متروك لتصميم المبرمج وتنفيذ الجزء من النظام الذي يواجه الخطأ.

يوفر هذا النهج للمطورين الأفراد مرونة عالية، ولكنه يُنشئ خطراً كبيراً ذلك بأن الأداء العام للنظام سوف لن يفي بمتطلباته من حيث الصحة أو المتانة (المزيد عن ذلك في لحظة).

اعتمادا على كيفية انهاء المطوّرين لمعالجة اخطاء محدّدة، فإن هذا النهج لديه القدرة كذلك على نشر شفرة واجهة المستخدم في أرجاء النظام، مما يعرّض البرنامج لجميع المشاكل المتعلقة بعرض رسائل الخطأ.

إيقاف التشغيل بعض الأنظمة تقوم بالتوقّف كلّما اكتشفت وجود خطأ. هذا النهج مفيد في التطبيقات الحسّاسة لموضوع السلامة. على سبيل المثال، إذا كان البرنامج الذي يتحكم في معدّات الإشعاع لعلاج مرضى السرطان يتلقى بيانات إدخال سيئة لجرعة الإشعاع، فما هي أفضل استجابة للتعامل مع الأخطاء؟

هل ينبغي أن تستخدم القيمة نفسها التي كانت عليها في المرّة الأخيرة؟ هل ينبغي أن تستخدم أقرب قيمة مقبولة؟ هل ينبغي أن تستخدم قيمة محايدة؟ في هذه الحالة، الإيقاف هو الخيار الأفضل. سنفضّل كثيراً إعادة تشغيل الجهاز على التعرض لخطر تسليم الجرعة الخاطئة.

يمكن استخدام نهج مشابه لتحسين الأمن في مايكروسوفت ويندوز. افتراضيّاً، يستمر تشغيل ويندوز حتى عندما يكون سجل الأمان ممتلئا. ولكن يمكنك ضبط ويندوز لإيقاف المخدّم إذا امتلئ سجل الأمان، والذي من الممكن أن يكون مناسباً في بيئة أمنية حرجة.

المتانة مقابل التصحيح

كما تُظهر أمثلة لعبة الفيديو والأشعة السينيّة، فإنّ أسلوب معالجة الأخطاء الأكثر ملاءمةً يعتمد على نوع البرنامج الذي يحدث فيه الخطأ.

وتوضّح هذه الأمثلة أيضاً أن معالجة الأخطاء تُفضّل عموماً المزيد من الصوابية "الصحّة" أو المزيد من المتانة. يميل المطوّرون إلى استخدام هذه المصطلحات بشكل غير رسمي، ولكن، بالمعنى الدقيق للكلمة، تكون هذه المصطلحات على طرفي نقيض في المقياس من بعضها البعض.

فالصحّة تعني عدم إرجاع نتيجة غير دقيقة أبداً؛ عدم إرجاع أي نتيجة أفضل من إرجاع نتيجة غير دقيقة. بينما تعني المتانة أن تحاول دائما أن تفعل شيئا ما من شأنّه السماح للبرنامج بأن يبقى قيد التشغيل، حتى لو أدى ذلك إلى نتائج غير دقيقة في بعض الأحيان.

تميل التطبيقات ذات الحساسيّة للسلامة إلى تفضيل الصحّة على المتانة. فمن الأفضل عدم إرجاع أية نتيجة من إرجاع نتيجة خاطئة. تعتبر آلة الإشعاع مثال جيد لهذا المبدأ.

وتميل التطبيقات الاستهلاكية إلى تفضيل المتانة على الصحّة. أية نتيجة على الإطلاق هي عادةً أفضل من إغلاق البرنامج. يعرض معالج النصوص وورد الذي أستخدمه في بعض الأحيان جزءاً من سطر نصي في أسفل الشاشة. إذا اكتشفت هذه الحالة، هل أريد إيقاف تشغيل معالج النصوص؟ لا، أنا أعرف أنّه في المرة التالية التي أضغط فيها على زر صفحة للأعلى"Page Up" أو صفحة للأسفل"Page Down`". ستحدّث الشاشة وستعود إلى وضعها الطبيعي.

تضمين التعميم عالي المستوى لمعالجة الأخطاء



مع العديد من الخيارات، سيتوجب توخي الحذر في التعامل مع الوسطاء غير الصالحة بطرق ثابتة في كافة أنحاء البرنامج.

الطريقة التي تعالج بها الأخطاء تؤثّر على قدرة البرنامج على تلبية متطلبات متعلقة بالصحّة، والمتانة، وغيرها من الخصائص غير الوظيفيّة. إنّ قرار اختيار النهج العام للتعامل مع الوسطاء السيئة هو قرار هيكلي أو قرار تصميم عالي المستوى ويجب معالجته في أحد هذين المستويين.

بمجرد اتخاذ قرار حول النهج، تأكّد من أنّك ستتابع به باستمرار. إذا قرّرت اتباع معالجة أخطاء عالية المستوى للشفرة وفقط تقارير أخطاء منخفضة المستوى للشفرة، تأكّد من أنّ معالج الأخطاء عالية المستوى للشفرة تتم بالفعل!

بعض اللغات تتيح لك إمكانية تجاهل حقيقة أن التابع يعيد شفرة خطأ- في سي++، لا تحتاج للقيام بأي شيء للقيمة المرجعة للتابع- ولكن لا تتجاهل معلومات الخطأ! اختبر قيمة التابع المرجعة. تحقق من ذلك على أي حال، حتى ولو كنت لا تتوقع أن ينتج التابع خطأ مطلقاً.

النقطة الأساسيّة للبرمجة الوقائية تكمن في الحماية ضد الأخطاء غير المتوقعة.

وهذا المبدأ يعتبر صحيحاً بالنسبة لتوابع النظام كما هو الحال بالنسبة لتوابعك الخاصة. إذا لم يكن لديك هيكلة للمبادئ لعدم تحقق استدعاءات النظام للأخطاء، تفحص شفرات الخطأ بعد كل استدعاء. إذا اكتشفت خطأ ما، ضمّن رقم الخطأ ووصفه.

4.8 الاستثناءات

الاستثناءات هي وسائل محدّدة تحدد شفرة برمجية يمكن أن تُمرر عبر أخطاء أو أحداث استثنائية إلى الشفرة التي استدعتها. إذا واجهت إجرائية حالة غير متوقعة لا تعرف كيفية التعامل معها، فسترمي استثناء Exception، وترفع يديها من الموضوع وتترك الأمر إلى غيرها.

"لا أعرف ماذا أفعل حيال هذا- أنا بالتأكيد آمل أن شخصاً ما آخر سيساعدني لمعرفة كيفية التعامل معها"

الشفرة التي لا تستشعر لسياق خطأ ما يمكنها أن تعيد التحكم إلى أجزاء أخرى في النظام والتي من الممكن أن تمتلك قدرة أكبر على تفسير الخطأ وفعل شيء ما مفيد حياله.

ويمكن أيضاً استخدام الاستثناءات لتصويب منطق معقّد داخل امتداد واحد للشفرة، كمثال" إعادة الكتابة باستخدام try-finally" في القسم 3.17.

البنية الرئيسيّة للاستثناء هي تلك التي تستخدم خلالها الإجرائية لرمي كائن الاستثناء exception object. ستلتقط الاستثناء شفرة في إجرائيات أخرى في تسلسل الاستدعاء داخل كتلة try-catch.

تختلف اللغات الشائعة في كيفيّة تنفيذ الاستثناءات. يلخص الجدول 8-1 الاختلافات الرئيسيّة بين ثلاثة منها

الجدول 8-1 دعم اللغة الشائعة للاستثناءات

خاصية الاستثناء	سي ++	جافا	فيجوال بيسك
دعم *Try-catch*	نعم	نعم	نعم
دعم *Try-catch-finally*	لا	نعم	نعم
ما الذي يمكن رميه	كائن استثناء أو كائن مستمد من صف *Exception*؛ مؤشر كائن. مرجع الكائن؛ نوع البيانات مثل int أو string	كائن استثناء أو كائن مستمد من صف *Exception*.	كائن استثناء أو كائن مستمد من صف *Exception*.
تأثير الاستثناء غير المعالَج uncaught exception	إستدعاء *std::unexpected()*، والتي تستدعي افتراضياً *std::terminate()*، والتي تستدعي افتراضياً *abort()*	أنهاء مسار thread الاستثناء إذا كان الاستثناء استثناء مدقّق" checked exception"، لا تأثير إذا كان الاستثناء استثناء وقت التشغيل " runtime exception "	إنهاء البرنامج
يجب تعريف الاستثناءات المرمية في واجهة الصف	لا	نعم	لا

للاستثناءات خاصيّة مشتركة مع الوراثة[1]: استخدامها بحكمة، يمكن أن يقلل من التعقيد. واستخدامها بتهور يمكن أن يجعل الشفرة غالباً مستحيلة التتبع.

يحتوي هذا القسم على اقتراحات للاستفادة من فوائد الاستثناءات وتجنب الصعوبات التي غالباً ما ترتبط بها.

استخدم الاستثناءات *لإشعار أجزاء البرنامج الأخرى بالأخطاء التي لا ينبغي تجاهلها* وتعتبر الفائدة الأساسيّة للاستثناءات هي في إمكانية الإشارة إلى حالات الخطأ بطريقة لا يمكن تجاهلها (مايرز 1996).

تنشئ النهج الأخرى في التعامل مع الأخطاء احتمال انتشار حالة الخطأ من أساس شفرة غير مكتشفة. تلغي الاستثناءات هذا الاحتمال.

رمي الاستثناءات فقط للحالات التي تعتبر استثنائية بالفعل يجب أن تحفظ الاستثناءات للحالات التي تعتبر استثنائية حقاً-بكلمات أخرى، للحالات التي لا يمكن معالجتها بنشاطات الشفرة الأخرى. تستخدم الاستثناءات في ظروف مشابهة للتأكيدات `assertions`- ليس للأحداث التي تعتبر نادرة فحسب ولكن من أجل الأحداث التي يجب ألا تحدث.

تمثل الاستثناءات مقايضة بين طريقة قويّة للتعامل مع حالات استثنائية من جهة وزيادة التعقيد من جهة أخرى. تضعف الاستثناءات التغليف `Encapsulation` من خلال طلب الشفرة التي تستدعي الإجرائية لمعرفة أي استثناء يمكن أن يُرمى داخل الشفرة التي تم استدعاؤها.

مما يزيد من تعقيد الشفرة، وهذا ما لا يوافق ما يشير إليه الفصل 5"التصميم في البناء" للالتزام بتقنية البرمجيات الرئيسيّة: إدارة التعقيد.

لا تستخدم استثناء لتمرر الـ buck إذا كان بالإمكان التعامل محلياً مع حالة خطأ ما، تعامل معها محلياً. لا ترمي استثناء غير معالج `uncaught exception` في مقطع من الشفرة إذا كان يمكن التعامل معه محلياً.

تجنب رمي الاستثناءات في البواني `Constructors` **والمهدّمات** `Destructors` **إلاّ إذا تم التقاطهم في نفس المكان** تصبح القواعد الخاصّة بكيفية معالجة الاستثناءات أكثر تعقيداً وسرعة عندما يرمى الاستثناء في الباني أو الهادم. على سبيل المثال، لا يتم استدعاء الهادم في سي++ مالم يتم بناء الكائن بالكامل. مما يعني أنّه إذا رمت الشفرة داخل الباني استثناء، فلن يتم استدعاء الهادم، مما يمكن أن يؤدي إلى فقدان محتمل في الموارد (مايرز 1996، ستروستروب 1997). تطبق قواعد معقّدة مشابهة على الاستثناءات داخل الهوادم.

[1] البرامج التي تستخدم الاستثناءات كجزء من عملية المعالجة العاديّة تعاني من جميع مشاكل إمكانية القراءة والصيانة لشفرة `spaghetti` الكلاسيكية.

من الممكن أن يقول محامو اللغة أنّ تذكر قاعدة كهذه "تافه"، لكن المبرمجين واللذين هم بشر في النهاية يجدون صعوبة في تذكرها.

إنّها ببساطة أفضل نشاط برمجي لتجنب التعقيد الإضافي كإنشاء شفرة من خلال عدم كتابة هذا النوع من الشفرات في المقام الأول.

رمي الاستثناءات في المستوى الصحيح للتجريد Abstraction[1] ينبغي أن تقدم الإجرائية تجريداً متسقاً في واجهتها. وكذلك الأمر بالنسبة للصف. الاستثناءات المرميّة هي جزء من واجهة الإجرائية، تماماً مثل أنواع البيانات المحدّدة. عندما تختار تمرير استثناء إلى المستدعي، تأكد من أن مستوى تجريد الاستثناء متوافق مع تجريد واجهة الإجرائية. وإليك هذا المثال لما لا يجب عليك فعله:

شفرة مرعبة

مثال جافا سيء لصف يرمي استثناء في مستوى تجريد غير ملائم.

```
class Employee {
   ...
   public TaxId GetTaxId() throws EOFException {                    ...
   }
   ...
}
```

هنا التصريح للاستثناء الذي يقع في مستوى تجريد غير ملائم

تجتاز شفرة الـ GetTaxId() المستوى المنخفض للاستثناء_EOFException عائدة إلى المستدعي. وهي لا تأخذ ملكية الاستثناء نفسه; إنّها تعرض بعض التفاصيل حول كيفية تنفيذه من خلال تمرير استثناء المستوى المنخفض إلى المستدعي.

هذه الفعالية تجمع شفرة زبون الإجرائية وليس لشفرة الصف Employee ولكن للشفرة تحت صف الـ Employee التي ترمي استثناء EOFException. تم كسر التغليف EOFException، وتبدأ المرونة الذهنية بالانخفاض.

عوضاً عن ذلك، يجب أن تمرر شفرة GetTaxId() استثناء يتطابق مع واجهة الصف الذي يكون جزءاً منها، كما يلي:

مثال جافا جيد لصف يرمي استثناء في مستوى ملائم من تجريد.

```
class Employee {
...
public TaxId GetTaxId() throws EmployeeDataNotAvailable {
...
}
...
}
```

هنا تصريح عن الاستثناء الذي يشارك في مستوى ملائم للتجريد

[1] **إشارة مرجعية** لمزيد من المعلومات للحفاظ على واجهة تجريد متسقة، انظر "التجريد الجيّد" في المقطع 2.6.

شفرة معالجة الاستثناء داخل GetTaxId() ربما فقط ستربط استثناء /ـ/ io_disk_not_ready مع استثناء EmployeeDataNotAvailable، والذي يعتبر جيداً لأنّه كفيل بالحفاظ على واجهة التجريد.

ضمّن في رسالة الاستثناء كل المعلومات التي أدّت إلى الاستثناء يحدث كل استثناء في ظروف معيّنة تكتشف في الوقت الذي ترمي فيه الشفرة الاستثناء. هذه المعلومات ثمينة بالنسبة للشخص الذي يقرأ رسالة الاستثناء. كن متأكداً من أنّ الرسالة تحتوي المعلومات المطلوبة لفهم سبب رمي الاستثناء.

إذا تم رمي الاستثناء بسبب خطأ فهرس مصفوفة، تأكد من تضمين الرسالة الحدود العليا والدنيا للمصفوفة وقيمة الفهرس غير النظامية.

تجنّب كتل catch الفارغة أنّه من المغري في بعض الأحيان تجاوز الاستثناء الذي لا تعرف ما الذي يجب أن تفعله تجاهه، مثل هذا:



مثال جافا سيء لتجاهل استثناء

```
try {
...
// الكثير من الشفرة
...
} catch ( AnException exception ) {
}
```

يقول مثل هذا النهج أنّه إما أنّ الشفرة داخل كتلة try خاطئة لأنها رفعت خطأ دون سبب، أو أن الشفرة داخل كتلة catch خاطئة لأنها لم تعالج استثناء صحيح. حدّد جزر المشكلة، ثم أصلح إما الكتلة try أو الكتلة catch.

من الممكن أن تجد أحياناً حالات نادرة بحيث أن الاستثناء في مستوى منخفض لا يمثل حقاً استثناء في مستوى تجريد الإجرائية المستدعية. إذا كان الحال كذلك، وثّق على الأقل لماذا تخصّص كتلة catch *فارغة*. يمكنك "توثيق" تلك الحالة في التعليقات أو من خلال تدوين الرسالة في ملف، كالتالي:

مثال جافا جيد لتجاهل استثناء

```
try {
...
// الكثير من الشفرة
...
} catch ( AnException exception ) {
LogError( "Unexpected exception" );
}
```

اعرف الاستثناءات الذي ترميها مكتبة شيفرتك إذا كنت تعمل على لغة لا تتطلب إجرائية أو صف لتعريف الاستثناء الذي ترميه، فتأكد من معرفة ماهي الاستثناءات المرمية من قبل أي مكتبة شفرة تستخدمها، الفشل في التقاط استثناء ناتج عن شفرة مكتبة سيعطّل برنامجك بسرعة فشل التقاط الاستثناء الذي ولدته بنفسك.

إذا لم توثّق مكتبة شفرة المكتبة الاستثناء المرمي، أنشئ شفرة نموذج أولي وأزل الاستثناءات.

التفكير في بناء مُخبر استثناء مركزي (centralized exception reporter) أحد النُهج المتبعة لضمان الاتساق في التعامل مع الاستثناءات هي استخدام مخبر استثناء مركزي. centralized exception reporter. يوفر مخبر الاستثناء المركزي مخزن مركزي لمعرفة ماهي أنواع الاستثناءات الموجودة، وكيفية التعامل مع كل استثناء، وتهيئة رسائل الاستثناء، وما إلى ذلك.

هنا مثال لمعالج استثناء بسيط يقوم ببساطة بطباعة رسالة تشخيص[1]:

مثال فيجوال بيسك لمخبر الاستثناء المركزي، جزء 1

```
Sub ReportException( _
ByVal className, _
ByVal thisException As Exception _
)
Dim message As String
Dim caption As String
message = "Exception: " & thisException.Message & "." & ControlChars.CrLf & _
"Class: " & className & ControlChars.CrLf & _
"Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
caption = "Exception"
MessageBox.Show( message, caption, MessageBoxButtons.OK, _
MessageBoxIcon.Exclamation )
End Sub
```

يمكنك استخدام معالج الاستثناء العام هذا مع شفرة كالتالي:

مثال فيجوال بيسك لمخبر الاستثناء المركزي، جزء 2

```
Try
...
Catch exceptionObject As Exception
ReportException( CLASS_NAME, exceptionObject )
End Try
```

[1] **مزيد من القراءة** للحصول على شرح أكثر تفصيلا لهذه التقنية، انظر المعايير العملية لمايكروسوفت فيجوال بيسك دوت نت (فوكسال2003)

الشفرة في نسخة ReportException() هذه بسيطة. يمكنك في التطبيق الحقيقي، جعل الشفرة بسيطة أو مفصّلة حسب الحاجة لتلبية احتياجات معالجة استثناءك.

إذا قررت بناء مُخبر استثناء مركزي، تأكّد من النظر في القضايا العامة التي ينطوي عليها التعامل مع الأخطاء المركزية، والتي تتم مناقشتها في "استدعاء معالجة خطأ الإجرائية/الكائن" في المقطع 3.8.

وحّد طريقة استخدام مشروعك للاستثناءات

للحفاظ على معالجة الأخطاء مرنة قدر الإمكان، يمكنك توحيد استخدامك للاستثناءات بعدة طرق:

- إذا كنت تعمل على لغة مثل سي++ والتي تسمح لك برمي مجموعة متنوعة من الكائنات، والبيانات، والمؤشرات، وحّد ما سوف يرمي تحديداً. للتوافق مع اللغات الأخرى، خذ بعين الاعتبار رمي فقط الكائنات المستمدة من الصف الأب Exception.
- فكّر في إنشاء صف استثناء خاص بمشروعك، والذي يمكن أن يكون بمثابة الصف الأب لكل الاستثناءات المرمية في مشروعك. وهذا من شأنه دعم مركزية وتوحيد التوثيق logging، وتقرير الأخطاء error reporting، وهلمّ جراً.
- عرّف الظروف المحدّدة التي تحدد أي شفرة مسموح لها باستخدام قاعدة throwcatch لتنفيذ معالجة الخطأ محلياً.
- عرّف الظروف المحدّدة التي تحدد أي شفرة مسموح لها برمي استثناء لن يتم التعامل معه محلياً.
- حدّد فيما إذا كان سيتم استخدام مخبر استثناء مركزي.
- وضح إذا كان استخدم الاستثناءات مسموحا في البواني والهوادم.

التفكير في بدائل الاستثناءات[1] دعمت العديد من لغات البرمجة الاستثناءات من 5-10 سنوات أو أكثر، ولكن قدر صغير من الحكمة التقليدية نتجت حول كيفية استخدامهم بأمان.

يستخدم بعض المبرمجون الاستثناءات لمعالجة الأخطاء فقط لأن لغاتهم توفر لهم آلية معينة لمعالجة الأخطاء. يجب عليك التفكير دائما في مجموعة كاملة من بدائل معالجة الأخطاء: معالجة الخطأ محلياً، أو بث الخطأ باستخدام شفرة الخطأ، أو تسجيل معلومات التصحيح في ملف، أو إيقاف تشغيل النظام، أو استخدام نهج أخرى.

إنّ معالجة الاخطاء باستخدام الاستثناءات فقط لكون لغتك توفر لك معالجة استثناءات هو مثال تقليدي للبرمجة في لغة بدلا من البرمجة إلى لغة. (للمزيد من المعلومات حول هذا الاختلاف، انظر القسم 3.4،"مكانك

[1] **إشارة مرجعية** للعديد من النهج البديلة لمعالجة الأخطاء، انظر القسم 3.8، "تقنيات معالجة الأخطاء"، في وقت سابق من هذا الفصل.

على الموجة التكنولوجية،" والقسم 4.34، "برمج إلى لغتك، وليس فيها". Program into Your Language، Not in It)

أخيراً، ادرس ما إذا كان البرنامج حقا يحتاج الى معالجة الاستثناءات، نقطة. كما أشار بجارن ستروستروب, أفضل استجابة أحيانا لأخطاء زمن التشغيل الخطيرة هي بتحرير كل المصادر المكتسبة والإلغاء.

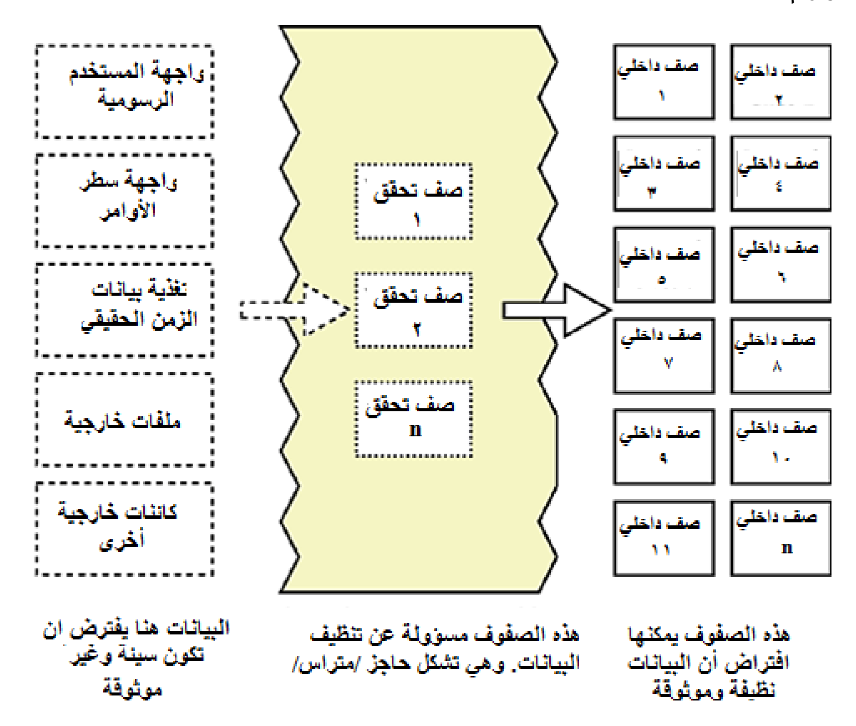
اسمح للمستخدم بإعادة تشغيل البرنامج مع المدخلات الصحيحة (ستروستروب 1997) .

5.8 حصّن برنامج لاحتواء الضرر الناجم عن الأخطاء

المتاريس Barricades هي استراتيجية احتواء الضرر. والسبب مشابه لامتلاك حجرات منعزلة في هيكل السفينة. إذا دخلت السفينة إلى جبل جليدي وفتح الدفع الهيكل، تُغلق هذه المقصورة دون أن تتأثر بقية أجزاء السفينة.

كما أنّها مشابهةٌ أيضاً لجدران النار في البناء. تمنع الجدران الناريّة للبناء النار من الانتشار من جزء من البناء إلى جزء آخر. (المتاريس والتي كان يطلق عليها جدران النار"firewalls", لكن الآن مصطلح جدار النار أصبح يشير عادة إلى حظر تدفق الشبكات المعادية.)

وأحد طرق التحصين لأغراض البرمجة الوقائية هي تعيين واجهات محدّدة كحدود لمناطق "أمنة". تحقق من اجتياز البيانات لحدود منطقة الأمان للتأكد من صحتها، وتستجيب بإدراك إذا كانت البيانات غير صالحة. يوضّح الشكل 8-2 هذا المفهوم



الشكل 8-2 تعريف بعض أجزاء البرنامج التي تعمل مع البيانات غير النظيفة والبعض الذي يعمل مع البيانات النظيفة يمكن أن يكون وسيلة فعّالة لتقليل حجم الشفرة الكبير الذي يتحمل المسؤولية عن التحقق من البيانات السيئة.

ويمكن استخدام هذا النهج نفسه على مستوى الصف. وتفترض المناهج لعامّة للصف أنّ البيانات غير آمنة، وهي مسؤولة عن فحص البيانات وتنظيفها.

وبمجرد قبول البيانات من قبل المناهج العامّة للصف، يمكن للصفوف الخاصة للصف أن تفترض أن البيانات آمنة.

هناك طريقة أخرى للتفكير في هذا النهج وهي تقنية غرفة العمليات. حيث يتم تعقيم البيانات قبل السماح لها بدخول غرفة العمليات. أي شيء في غرفة العمليات يفترض أن يكون آمناً.

قرار التصميم الرئيسي هو تقرير ما الذي يجب وضعه داخل غرفة العمليات، وما الذي سيبقى خارجاً، وأين توضع الأبواب التي تقدرها الإجرائية لتكون داخل منطقة الأمان. وما سيكون خارجها، ومن سيعقّم البيانات.

عادةً أسهل طريقة للقيام بذلك هي بتعقيم البيانات الخارجية عند وصولها، ولكن

غالباً ما تحتاج البيانات إلى التعقيم على أكثر من مستوى واحد، لذلك فإن عدّة مستويات من التعقيم مطلوبة في بعض الأحيان.

حوّل بيانات الإدخال إلى النوع المناسب عند الإدخال غالباً ما يصل الدخل بصيغة سلسلة محرفية أو رقم. سوف تُردّ القيمة إلى نوع منطقي Boolean مثل "نعم" أو "لا". وأحياناً ستُردّ إلى نوع تعداد مثل Color_Red, Color_Green و Color_Blue. إنّ حمل بيانات من نوع مشكوك فيه لأي وقت في برنامج يزيد من التعقيد كما يزيد من فرصة أن يتسبب شخص ما بانهيار برنامجك عن طريق ادخال لون مثل"Yes". حوّل بيانات الإدخال إلى الشكل المناسب في أقرب وقت بعد إدخالها.

العلاقة بين الحواجز "المتاريس" والتأكيدات

استخدام الحواجز Barricades يجعل التمييز بين التأكيدات Assertions ومعالجة الخطأ أوضح clean-cut ينبغي على الإجرائيات الموجودة خارج الحاجز استخدام معالجة أخطاء لأنّه من غير الآمن إجراء أية افتراضات حول البيانات. ينبغي أن تستخدم الإجرائيات داخل الحاجز التأكيدات لأن البيانات الممرّرة إليها من المفترض أن يتم تعقيمها قبل مرورها عبر الحاجز.

إذا اكتشفت إحدى الإجرائيات داخل الحاجز بيانات سيئة، فذلك خطأ في البرنامج أكثر منه خطأ في البيانات.

يوضح استخدام الحواجز أيضاً قيمة اتخاذ قرار على مستوى الهيكلة في كيفية التعامل مع الأخطاء. اتخاذ قرار أي شفرة داخل الحاجز وأيها خارجه هو قرار على مستوى الهيكلة.

6.8 وسائل التصحيح

إنّ استخدام وسائل التصحيح هو مفهوم رئيسي آخر للبرمجة الوقائية، والتي يمكن أن تكون حليفاً قوياً في عملية الكشف السريع عن الأخطاء.

لا تطبق تلقائياً قيود الانتاج على نسخة التطوير

إنّ النقطة العمياء الشائعة للمبرمج هي افتراض أن القيود المفروضة على إنتاج البرمجيات تنطبق على نسخة التطوير[1]. حيث يجب على نسخة الانتاج "النهائية" أن تعمل بشكل سريع. أما نسخة التطوير فيمكن أن تعمل ببطء. ويجب على نسخة الانتاج أن تكون بخيلة بالنسبة للموارد. في حين أنه قد يُسمح لنسخة التطوير باستخدام الموارد بإسراف. كما لا يجب على نسخة الإنتاج أن تعرض العمليات الخطرة على المستخدم. ولكن من الممكن أن تملك نسخة التطوير عمليات إضافية، يمكنك استخدامها بدون شبكة أمان.

استخدمت أحد البرامج التي عملت عليها بشكل واسع القائمة المرتبطة بشكل رباعي. كان النص البرمجي للقائمة المرتبطة معرّض للخطأ، وقد مالت القائمة المرتبطة إلى التلف. فقمت بإضافة خيار القائمة للتحقق من سلامة القائمة المرتبطة.

في نمط التصحيح، يحتوي مايكروسوفت وورد على شفرة في وقت الخمول، تتحقق من سلامة الكائن Document كل بضعة ثواني. يساعد هذا على كشف تلف البيانات بسرعة، ويجعل من السهل تشخيص الأخطاء.



كُن مستعد للمقايضة بين استخدام الموارد والسرعة خلال التطوير عوضاً عن الأدوات المدمجة، التي يمكن أن تجعل التطوير أكثر سلاسة.

أدخل وسائل التصحيح في وقت مبكّر

كلما قمت بإدخال وسائل التصحيح أبكر، كلما ساعدَتْ هذه الأدوات أكثر. بشكل نموذجي، لن تسعى إلى كتابة وسيلة تصحيح حتى تكون قد عانيت من نفس المشكلة عدّة مرات. إذا قمت بكتابة وسيلة التصحيح بعد أول مرة، على كل حال، أو استخدمت واحدة من الأدوات من مشروع سابق، سوف يساعدك هذا في المشروع.

استخدم البرمجة الهجومية

[1] **قراءة متعمقة**: للمزيد حول استخدام شفرة التصحيح لدعم البرمجة الوقائية، انظر "كتابة شفرة صلبة " (ماغواير 1993).

يجب أن يتم معالجة حالات الاستثناء بطريقة تجعلهم واضحين خلال التطوير وقابلين للاسترداد عند تشغيل شفرة الانتاج[1]. أشار مايكل هاورد وديفيد ليبلانك إلى هذه الطريقة كـ "برمجة وقائية" (هوارد و ليبلانك 2003).

افترض أنه لديك عبارة case، التي تتوقع أن تعالج خمس أنواع فقط من الأحداث. خلال التطوير، يجب أن تُستخدم الحالة الافتراضية لتوليد رسالة تحذير تقول " مهلاً! هناك حالة أخرى هنا! أصلح البرنامج "، أما خلال الانتاج، على كل حال، يجب على الحالة الافتراضية أن تقوم بشيء ما أكثر رشاقةً، ككتابة رسالة إلى ملف سجل الأخطاء.

فيما يلي، بعض الطرق التي يمكنك بها برمجة البرامج بشكل هجومي[2]:

- تأكّد من مصــادقات إجهاض البرنامج. لا تســمح للمبرمجين بالدخول في عادة الضــغط فقط على زر الدخول لتجاوز مشكلة معروفة. اجعل المشكلة مؤلمة بما فيه الكفاية حتى يتم إصلاحها.
- املأ بشكل كامل أية ذاكرة مخصّصة، وبذلك يمكنك الكشف عن أخطاء تخصيص الذاكرة.
- املأ بشكل كامل أي ملف أو التدفقات المخصصة لتطرد أيّة أخطاء تنسيق الملف.
- تأكّد من أنّ البرنامج، في كل حالة افتراضــية أو أية حالة أخرى، يفشــل بشــكل صــعب (إجهاضــات البرنامج)، أو أنه من المستحيل التغاضي عن ذلك.
- املأ كائن ببيانات غير مرغوب فيها، فقط قبل حذفه.
- أعد البرنامج إلى ملف تسجيل أخطاء عن طريق البريد الإلكتروني، بهذا تستطيع أن ترى أنواع الأخطاء التي تحدث في البرمجية المُصدَرة، إذا كان ذلك مناسباً لنوع البرمجيات التي تقوم بتطويرها.

أحياناً أفضل دفاع هو هجوم جيد. ولذلك فإن الفشل بشكل كبير أثناء التطوير، يجعل الفشل قليلاً خلال الانتاج.

خطة لإزالة أدوات التصحيح

إذا كنت تكتب شفرة لاستخدامك الخاص، من الممكن أن تقرّر ترك كل الشفرة الخاصة بعملية التصحيح في البرنامج. وإذا كنت تكتب شفرة للاستخدام التجاري، ضريبة الأداء في السرعة والحجم يمكن أن تكون باهظة. عندها خطط لتجنب خلط الشفرة الخاصة بعملية التصحيح داخل وخارج البرنامج. فيما يلي عدة طرق للقيام بذلك:

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول معالجة الحالات غير المتوقعة، انظر " نصائح لاستخدام عبارات الحالة" في القسم 2.15

[2] عادة ما يؤدي البرنامج الميت إلى أضرار أقل بكثير من البرنامج المعطل. -اندي هانت وديف توماس

استخدم أدوات تحكم بالنسخة وأدوات بناء مثل ant و make.[1] تستطيع أدوات التحكم بالنسخة بناء عدة نسخ من البرنامج من نفس ملفات الشفرة المصدر. في وضع التطوير، يمكن إعداد أدوات البناء لتضمين كل النص البرمجي الخاص بعملية التصحيح. وفي وضع الانتاج، يمكن إعداد أدوات البناء لاستبعاد أي نص برمجي خاص بعملية التصحيح، لا تريده في النسخة التجارية من البرنامج.

استخدم المعالج التمهيدي المضمّن. إذا كان لدى البيئة البرمجية التي تعمل فيها معالج تمهيدي- كما لدى لغة البرمجة سي++- عندها يمكنك تضمين أو استبعاد الشفرة الخاصة بعملية التصحيح فقط بنقرة على مفتاح التحويل البرمجي. يمكنك استخدام المعالج التمهيدي بشكل مباشر أو عن طريق كتابة مُسجل "ماكرو"، يعمل مع تعاريف المعالج التمهيدي. فيما يلي مثال عن كتابة نص برمجي باستخدام المعالج التمهيدي بشكل مباشر:

```
#define DEBUG
...
 #if defined( DEBUG )
// شفرة تصحيح الأخطاء
...
#endif
```

لتضمين شيفرة التصويب، استخدم #define لتعريف الرمز DEBUG. لاستبعاد شفرة التصحيح لا تعرّف DEBUG.

لدى هذا الموضوع العديد من الاختلافات. بدلاً من تعريف DEBUGفقط، يمكن أن تسند له قيمة ومن ثم تختبر هذه القيمة بدلاً من اختبار ما إذا كانت معرّفة. بهذه الطريقة يمكنك التغيير بين عدّة مستويات من شفرة التصحيح. من الممكن أن ترغب بوجود شفرة تصحيح معينة في برنامجك بشكل دائم، عندها يمكنك أن تحيطها بعبارة من الشكل `#if DEBUG > 0`. من الممكن أن تكون شفرة تصحيح أخرى فقط لأغراض خاصّة، عندها يمكنك أن تحيطها بعبارة من الشكل

```
#if DEBUG == POINTER_ERROR
```

في أماكن أخرى، من الممكن أن ترغب بإعداد مستويات من التصحيح، عندها يمكنك استخدام عبارات من الشكل `#if DEBUG > LEVEL_A`.

إذا كنت لا ترغب بانتشار تعريفات `#if defined()` في شفرتك، عندها يمكنك كتابة مسجل "ماكرو" معالج تمهيدي للقيام بنفس المهمة. فيما يلي مثال عن ذلك:

مثال بلغة البرمجة سي++ عن استخدام ماكرو المعالج التمهيدي للتحكم بشفرة التصحيح.

```
#define DEBUG
#if defined( DEBUG )
#define DebugCode( code_fragment ) { code_fragment }
```

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول التحكم بالنسخة، انظر المقطع 2.28 "إدارة التكوين(الإعداد)"

```
#else
#define DebugCode( code_fragment )
#endif
...
DebugCode(
statement 1;
statement 2;
...
statement n;
);
 ...
```

هذا النص البرمجي يتم تضمينه أو استبعاده وذلك تبعاً لحالة DEBUG معرّفة أو لا.

كما في المثال الأول لاستخدام المعالج التمهيدي، يمكن تبديل هذه التقنية بمجموعة مختلفة من الطرق، التي تجعل منها أكثر تطوراً من التضمين الكامل لشفرة التصحيح أو الاستبعاد الكامل لها.

اكتب المعالج التمهيدي الخاص بك[1]. إذا لم تتضمن لغة البرمجة معالج تمهيدي، فمن السهل كتابة واحد لتضمين أو استبعاد شفرة التصحيح. قُم بوضع اتفاقية لتصميم شفرة التصحيح، واكتب المترجم التمهيدي لمتابعة هذه الاتفاقية. بالنسبة للمثال، في لغة البرمجة جافا عليك أن تكتب مترجم تمهيدي لتمثيل المفاتيح الأساسية DEBUG *//#BEGIN* و *//#END DEBUG* . اكتب شفرة لاستدعاء المعالج التمهيدي، ومن ثم ترجم النص البرمجي المُعالج. عندها ستوفر الوقت، ولن تقوم بشكل خاطئ بترجمة الشفرة غير المُعالجة مسبقاً.

استخدم إجرائيات التصحيح الصغيرة (Debugging stubs***)***[2]. في كثير من الحالات، تستطيع أن تستدعي إجرائية للقيام بفحوص التصحيح. أثناء التطوير، يمكن أن تنفذ الإجرائية العديد من العمليات قبل عودة التحكم إلى المُستدعي. أما بالنسبة إلى وضع الانتاج، يمكنك أن تستبدل الإجرائية المُجمعة بإجرائية أصغر stub، التي تقوم فقط بإعادة التحكم فوراً إلى المُستدعي، أو التي تنفذ بضع عمليات سريعة قبل إعادة التحكم. لا يترتب على هذه الطريقة سوى ضريبة أداء صغيرة، وهي عبارة عن حل أسرع من كتابة المعالج التمهيدي الخاص بك. حافظ على كل من نسختي التطوير والإنتاج للإجرائية، وبهذا تستطيع التبديل بينهما خلال التطوير والإنتاج المستقبليين.

من الممكن أن تبدأ بتصميم إجرائية تتفحص المؤشرات التي تُمرر لها:

[1] **إشارة مرجعية**: لمزيد من المعلومات عن المعالجات التمهيدية، وللتوجه إلى مصادر، تحوي معلومات عن كتابة معالج تمهيدي بنفسك، انظر القسم 3.30 "المعالجات التمهيدية للماكرو".

[2] **إشارة مرجعية:** لمزيد من التفاصيل حول الـ stubs، انظر " بناء السقالات لاختبار الإجرائيات المفردة" في القسم 5.22.

مثال بلغة البرمجة سي++ عن إجرائية تستخدم إجرائية التصحيح المختصرة Debugging Stub

```
void DoSomething(
SOME_TYPE *pointer;
...
){
// التحقق من الوسطاء التي تم تمريرها
CheckPointer( pointer );
...
}
```

يستدعي هذا السطر الإجرائية لتفحص المؤشرات.

خلال التطوير، ستنفذ الإجرائية *CheckPointer ()* الفحص الكامل على المؤشر. من الممكن أن يكون هذا بطيء ولكن فعّال، و من الممكن أن يبدو كالتالي:

مثال بلغة البرمجة سي++ عن إجرائية لتفحص المؤشرات خلال التطوير:

```
void CheckPointer( void *pointer ) {

// perform check 1--maybe check that it's not NULL
// perform check 2--maybe check that its dogtag is legitimate
// perform check 3--maybe check that what it points to isn't corrupted
...
// perform check n–
...
}
```

تتفحص هذه الإجرائية أي مؤشر يُمرر إليها. يمكن استخدامها خلال التطوير لأداء أكبر عدد ممكن الفحوص.

عندما تكون الشفرة جاهزة للإنتاج "للإصدار"، من الممكن أن ترغب بعدم تفحص جميع الأمور المرتبطة بهذا المؤشر. حيث يمكنك تبديل الإجرائية السابقة بهذه الإجرائية:

مثال بلغة البرمجة سي ++عن إجرائية لتفحص المؤشرات خلال الإنتاج

```
void CheckPointer( void *pointer ) {
// no code; just return to caller
}
```

تُعيد هذه الإجرائية مباشرةً للمُستدعي.

إنّ هذا ليس بملخص شامل لكل الطرق التي يمكن تخطيطها لإزالة وسائل التصحيح، ولكن ينبغي أن يكون كافياً لإعطائك فكرة عن بعض الأشياء التي ستعمل في بيئتك.

7.8 تحديد كمية البرمجة الوقائية المتروكة في الشفرة النهائية

إحدى مفارقات البرمجة الوقائية، هي أنّه خلال التطوير، قد ترغب في أن يكون الخطأ ملحوظاً- قد ترغب في أن يكون الخطأ شنيع بدلاً من خطر تجاهله. ولكن خلال الانتاج، ترغب في أن يكون الخطأ غير بارز قدر الإمكان، حتى يتحسن البرنامج أو يفشل بأمان. فيما يلي خطوط توجيهية لتحديد أية أدوات البرمجة الوقائية، تُترك في شفرة الانتاج، وأيُّها تُزال منها:

اترك في الشفرة ما يتحقق من الأخطاء الهامّة. حدّد أيّة مناطق من البرنامج يمكن أن تتحمل أخطاء غير مكتشفة، وأيّة مناطق لا يمكن أن تتحمل ذلك. على سبيل المثال، إذا كنت تكتب برنامج جداول البيانات، تستطيع أن تتحمل وجود أخطاء في مناطق تحديث الشاشة من البرنامج، لأن الضريبة الرئيسية للخطأ هي مجرد فوضى على الشاشة. ولكن لا تستطيع أن تتحمل الأخطاء في منطقة محرك الحساب، لأن أخطاء كهذه من الممكن أن تعطي نتائج غير صحيحة في جدول بيانات أحد الأشخاص. يُفضّل معظم المستخدمين عدم الترتيب في الشاشة على الحصول على حسابات خاطئة للضرائب والتدقيق باستخدام IRS.

أزل الشفرة التي تتحقق من الأخطاء الطفيفة. إذا كان لدى أحد الأخطاء أثر طفيف، قُم بإزالة الشفرة التي تتحقق من هذا الخطأ. في المثال السابق، من الممكن أن تحذف الشفرة التي تتفحص تحديث شاشة جدول البيانات. لا تعني الكلمة "حذف"، الحذف الفيزيائي للشفرة. بل تعني استخدام التحكم بالنسخة، أو مفاتيح المترجم التمهيدي، أو أية تقنية أخرى لترجمة البرنامج بدون استخدام هذا الجزء الخاص من الشفرة. إذا لم تكن المساحة مشكلة، فيمكنك تركه هذه الشفرة في شفرة الفحص، ولكن احتفظ بشكل مخفي بسجل رسائل الخطأ في ملف سجل الخطأ.

أزل الشفرة التي تُنتج عطل مفاجئ. كما ذكرت من قبل، خلال التطوير، عندما يكتشف برنامجك خطأ، قد ترغب في أن يكون هذا الخطأ قابل للملاحظة بحيث يمكنك أن تصلحه. غالباً أفضل طريقة لتنفيذ هذا الهدف هي بجعل البرنامج يطبع رسالة التصحيح ويتعطّل بشكل مفاجئ عندما يكتشف خطأ ما. وهذا مفيد حتى للأخطاء الثانوية.

أما خلال الإنتاج، يحتاج المستخدمون لفرصة لتخزين أعمالهم قبل أن يتعطل البرنامج بشكل مفاجئ، وهم على الأرجح مستعدون للتسامح مع بعض الأخطاء مقابل استمرار عمل البرنامج مدة كافية لهم للقيام بحفظ العمل. لا يقدّر المستخدمين أي شيء يؤدي إلى فقدان أعمالهم، بغض النظر كم سيساعد ذلك على التصحيح وفي نهاية المطاف يُحسّن جودة البرنامج. إذا احتوى برنامجك على شفرة تصحيح يمكن أن تُحدث ضياعاً في البيانات، فالأفضل إزالتها من نسخة الانتاج.

اترك الشفرة التي تساعد على التعطيل المفاجئ للبرنامج بأمان. إذا كان برنامجك يحتوي على شفرة تصحيح، تكتشف أخطاء محتملة قاتلة، عندها اترك هذه الشفرة، التي تسمح بالتعطيل المفاجئ للبرنامج بأمان. على سبيل المثال في البرنامج (مستكشف المريخ) Mars Pathfinder، ترك المهندسين بعض من شفرة التصحيح في التصميم. حيث حدث خطأ بعد هبوط المستكشف. ولكن باستخدام وسائل التصحيح المتروكة، تمكّن المهندسين في مختبر الدفع النفاث JPL من تشخيص المشكلة وتحميل الشفرة المنقحّة إلى المستكشف، وقد أتم المستكشف مهمته بشكل جيد (مارش 1999).

سجّل الأخطاء لموظفي الدعم الفني خذ بعين الاعتبار ترك رسائل التصحيح في شفرة الانتاج ولكن بتغيير سلوكها، هذا مناسب لنسخة الإنتاج. إذا كنت قد حملّت التعليمات البرمجية مع المصادقات التي توقف البرنامج أثناء التطوير، من الممكن أن تأخذ بعين الاعتبار إجرائية المصادقة لتسجيل الرسائل إلى ملف أثناء الانتاج، بدلاً من القضاء عليها تماماً.

تأكّد من أن رسائل الخطأ التي تتركها ودية إذا تركت رسائل خطأ داخلية في البرنامج، تأكّد من أنّها بلغة وديّة للمستخدم. في أحد برامجي الأوليّة، لقد تلقيت اتصال من مستخدمة، قالت أنّها تلقت عشر رسائل "لقد حصلت على تخصيص مؤشر سيء، نَفَس الكلب! "، لحسن الحظ لي، أنه كان لدى هذه المستخدمة حس الدعابة. الطريقة الفعّالة والشائعة هي بإعلام المستخدم بوجود "خطأ داخلي"، والقيام بإدراج عنوان بريد إلكتروني أو رقم هاتف يمكن للمستخدم استخدامه للإبلاغ عنه.

8.8 أن تكون مدافعا عن البرمجة الوقائية[1]

يولّد الكثير من البرمجة الوقائية المشاكل الخاصة بها. إذا كنت تتحقق من البيانات التي يتم تمريرها كوسطاء بكل طريقة يمكن تصورها، وفي كل مكان يمكن تصوره، سيصبح برنامجك كبير وبطيء. ما هو أسوأ من ذلك، نص برمجي إضافي لازم للبرمجة الوقائية يضيف تعقيداً إلى البرنامج. إنّ الشفرة المثبّتة للبرمجة الوقائية ليست خالية من العيوب، ومن المرجح أن تجد الخلل في شفرة البرمجة الوقائية كما في أيّة شفرة أخرى- على الأرجح أن يحدث هذا إذا كتبت هذه الشفرة بإهمال. فكّر في أي مكان عليك أن تكون وقائياً، وعيّن أولويات البرمجة الوقائية وفقاً لذلك.

[1] الكثير من أي شيء هو شيء سيء، ولكن الكثير من الويسكي هو فقط كافٍ-مارك توين.

لائحة اختبار: البرمجة الوقائية1

بشكل عام

- هل تحمي الإجرائية نفسها من بيانات الدخل السيئة؟
- هل استخدمت تأكيدات لتوثيق الافتراضات، وتضمين الشروط المسبقة والشروط اللاحقة؟
- هل تم استخدام التأكيدات فقط لتوثيق الشروط التي لا ينبغي أن تحدث أبداً؟
- هل تُحدّد الهيكلة أو التصميم عالي المستوى مجموعة مخصّصة من تقنيات معالجة الأخطاء؟
- هل تُحدّد الهيكلة أو التصميم عالي المستوى فيما إذا كان يجب على معالجة الأخطاء أن تدعم المتانة أو الصحّة؟
- هل تم إنشاء المتاريس لاحتواء الأثر المُدمر للخطأ ولإنقاص كمية الشفرة التي يجب أخذها بعين الاعتبار عند معالجة الخطأ؟
- هل تم استخدام وسائل التصحيح في الشفرة؟
- هل تم تثبيت وسائل التصحيح بطريقة، يمكن فيها تفعيل وفك تفعيل هذه الوسائل دون إحداث ضجة كبيرة؟
- هل كمية البرمجة الوقائية للشفرة مناسبة- لا كبيرة جداً ولا صغيرة جداً؟
- هل قمت باستخدام تقنيات البرمجة الهجومية لجعل الأخطاء صعبة للتغاضي عنها أثناء عملية التطوير؟

الاستثناءات

- هل عرّف مشروعك طريقة موحدة لمعالجة الاستثناءات؟
- هل أخذت بعين الاعتبار بدائل استخدام الاستثناء؟
- هل تمت معالجة الخطأ محلياً، بدلاً من رمي استثناء غير محلي، إذا كان هذا ممكن؟
- هل تتجنب الشفرة رمي الاستثناءات في البواني والمهدّمات؟
- هل كل الاستثناءات على المستويات المناسبة من التجريد للإجرائيات التي تقوم برميها؟
- هل يحتوي كل استثناء على كل المعلومات الخلفية للاستثناء ذو الصلة؟
- هل النص البرمجي متحرر من كتل (catch) الفارغة؟ (أو هل كتلة catch الفارغة مُناسبة فعلياً، هل هي موثّقة؟)

[1] cc2e.com/0868

قضايا الأمن

- هل الشــفرة التي تتحقق من بيانات الدخل الســيئة، تتحقق أيضــاً من محاولة طفح المخزن المؤقت، ومن حقن الـــSQL، ومن حقن الـــHTML، ومن طفح العدد الصحيح، وغيرها من المدخلات الخبيثة؟
- هل تم التحقق من جميع الشفرات التي تُعيد خطأ؟
- هل تم التقاط جميع الاستثناءات؟
- هل تتجنب رسائل الخطأ تزويد معلومات قد تساعد المهاجم على كسر النظام؟

مصادر إضافية

انظر إلى مصادر المرجعية التالية للبرمجة الوقائية:

الأمن

هوارد، مايكل، وديفيد ليبلانك. كتابة شفرة آمنة، الإصدار الثاني. ريدموند، واشنطن: صحافة مايكروسوفت، 2003.

Howard, Michael, and David LeBlanc. Writing Secure Code, 2d ed. Redmond, WA: Microsoft Press, 2003.

غطى هوارد و ليبلانك الآثار الأمنية المترتبة على الثقة بالمدخلات. فتح هذا الكتاب العين على عدد الطرق التي يمكن فيها انتهاك برنامج- بعضها يتعلق بممارسات البناء والكثير منها لا يتعلق بذلك. يغطي هذا الكتاب مجموعة كاملة من المتطلبات والتصميم وكتابة الشفرة، وقضايا الاختبار.

التأكيدات" المصادقات"

ماغوير، ستيف. كتابة الشفرة الصلبة. ريدموند، واشنطن: صحافة مايكروسوفت، 1993.

Maguire, Steve. Writing Solid Code. Redmond, WA: Microsoft Press, 1993.

يحتوي الفصل الثاني على مقالة ممتازة عن استخدام التأكيدات، بما فيها مجموعة متعددة ممتعة من الأمثلة عن التأكيدات في منتجات مايكروسوفت المعروفة بشكل جيد.

ستروستروب، بجارن. لغة البرمجة سي++، الإصدار الثالث. ريادينغ، ماساتشوستس: أديسون-ويسلي، 1997.

Stroustrup, Bjarne. The C++ Programming Language, 3d ed. Reading, MA: Addison-Wesley, 1997.

يصف المقطع 2.7.3.24 العديد من الاختلافات حول موضوع تنفيذ المصادقات في لغة البرمجة سي++، يما فيها العلاقة بين التأكيدات والشروط المسبقة والشروط اللاحقة.

ماير، برتراند. هيكلة البرمجيات غرضية التوجه، الإصدار الثاني. نيويورك، نيويورك: برنتيس هول بتر، 1997.

Meyer, Bertrand. Object-Oriented Software Construction, 2d ed. New York, NY: Prentice Hall PTR, 1997.

يحتوي هذا الكتاب على مناقشة نهائية عن الشروط المسبقة والشروط اللاحقة.

الاستثناءات

ماير، برتراند. بناء البرمجيات غرضية التوجه، الإصدار الثاني. نيويورك، نيويورك: برنتيس هول بتر، 1997.

Meyer, Bertrand. Object-Oriented Software Construction, 2d ed. New York, NY: Prentice Hall PTR, 1997.

يحتوي الفصل 12 على مناقشة مفصّلة عن معالجة الاستثناءات.

ستروستروب، بجارن. لغة البرمجة سي++، الإصدار الثالث. ريادينغ، ماساتشوستس: أديسون-ويسلي، 1997.

Stroustrup, Bjarne. The C++ Programming Language, 3d ed. Reading, MA: Addison-Wesley, 1997.

يحتوي الفصل 14 على مناقشة مفصّلة لمعالجة الاستثناءات في لغة البرمجة سي++. يحتوي المقطع 11.14 على ملخص ممتاز لـ 21 نصيحة لمعالجة الاستثناءات في سي++.

مايرز، سكوت. لغة البرمجة سي++ أكثر فعالية: 35 طريقة جديدة لتحسين برامجك وتصميماتك. ريادينغ، ماساتشوستس: أديسون-ويسلي، 1996.

Meyers, Scott. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley, 1996.

تصف البنود 9-15 العديد من الفروق الدقيقة في معالجة الاستثناءات في لغة البرمجة سي++.

أرنولد، كين، جيمس جوسلينج، وديفيد هولمز. لغة البرمجة جافا، الإصدار الثالث. بوستون، ماساتشوستس: أديسون-ويسلي، 2000.

Arnold, Ken, James Gosling, and David Holmes. The Java Programming Language, 3d ed. Boston, MA: Addison-Wesley, 2000.

يحتوي الفصل 8 على مناقشة لمعالجة الاستثناءات في لغة البرمجة جافا.

بلوش، جوشوا. دليل لغة البرمجة جافا الفعّالة. بوستون، ماساتشوستس: أديسون-ويسلي، 2001.

Bloch, Joshua. Effective Java Programming Language Guide. Boston, MA: Addison-Wesley, 2001.

تصف البنود 39-47 الفروق البسيطة لمعالجة الاستثناءات في لغة البرمجة جافا.

فوكسال، جيمس. معايير عملية للغة البرمجة مايكروسوفت فيجول بيسك دوت نيت. ريدموند، واشنطون: صحافة ميكروسوفت، 2003.

Foxall, James. Practical Standards for Microsoft Visual Basic .NET. Redmond, WA: Microsoft Press, 2003.

يصف الفصل العاشر معالجة الاستثناءات في لغة البرمجة فيجول بيسك.

نقاط مفتاحية

- يجب أن تتعامل شفرة الانتاج مع الأخطاء بطريقة أكثر تطوراً من "دخل سيء، خرج سيء"
- تجعل تقنيات البرمجة الوقائية الأخطاء سهلة الكشف، وسهلة الاصلاح، وأقل ضرراً على شفرة الانتاج.
- تستطيع التأكيدات المساعدة في الكشف عن الأخطاء بشكل مبكر، وخاصّة في الانظمة الكبيرة، وفي الأنظمة عالية الموثوقية، وقواعد الشفرة سريعة التغيير.
- القرار حول كيفية التعامل مع المدخلات السيئة هو قرار معالجة الخطأ الرئيسي وقرار التصميم العالي المستوى الرئيسي.
- تؤمن الاستثناءات وسائل معالجة الأخطاء، التي تعمل في بعد مختلف عن المسار الطبيعي للشفرة. إنها أدوات قيمة إضافية لصندوق أدوات البرمجة الذكيّة للمبرمج، عندما يقوم باستخدامها بحذر، ينبغي موازنتها مع تقنيات معالجة الأخطاء الأخرى.
- القيود المطبّقة على نظام الانتاج ليس بالضرورة أن تُطبق على نسخة التطوير. يمكنك استخدام ذلك لصالحك، حيث يساعد إضافة نص برمجي إلى نسخة التطوير على استبعاد الأخطاء بسرعة.

الفصل 9

عملية برمجة الشفرة الزائفة

المحتويات[1]

- 1.9 تلخيص لخطوات بناء الصفوف والإجرائيات
- 2.9 الشفرة الزائفة من أجل الإيجابيات
- 3.9 بناء الإجرائيات باستخدام عملية برمجة الشفرة الزائفة
- 4.9 بدائل عملية برمجة الشفرة الزائفة

مواضيع ذات صلة

- تشكيل الصفوف عالية الجودة: الفصل 6
- خصائص الإجرائيات عالية الجودة: الفصل 7
- التصميم في البناء: الفصل 5
- أسلوب التعليق: الفصل 32

على الرغم من أنه بإمكانك أن ترى كامل هذا الكتاب كوصف موسّع لعملية البرمجة من أجل إنشاء الصفوف والإجرائيات، إلا أنّ هذا الفصل سيشرح الخطوات في هذا السياق. يُركز هذا الفصل على البرمجة في مجال صغير – على خطوات خاصّة لبناء الصف المستقل واجراءاته، حيث تُعد هذه الخطوات حرجة لكل المشاريع من كل الاحجام. يصف هذا الفصل أيضاً عملية برمجة الشفرة الزائفة - Pseudocode) Programming Process PPP)، التي تقلّل من العمل المطلوب خلال عملية التصميم وعملية التوثيق، وُحسّن جودة كلا العمليتين.

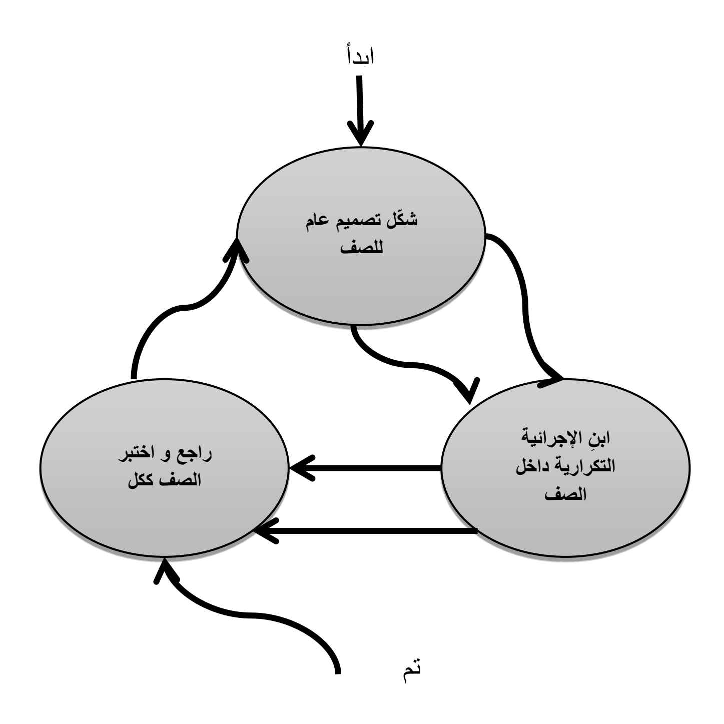
إذا كنت مبرمجاً خبيراً، فيمكنك فقط تصفح هذا الفصل، ولكن اطلع على تلخيص الخطوات وراجع النصائح لبناء الإجرائيات باستخدام عملية برمجة الشفرة الزائفة، في القسم 9-3 يستغل بعض المبرمجين الاستطاعة الكاملة لهذه العملية، والتالي يكسبون العديد من الفوائد.

[1] cc2e. com/0936

إنّ عملية برمجة الشفرة الزائفة ليست الإجرائية الوحيدة لإنشاء الصفوف والإجرائيات. يصف القسم 9-4، في نهاية هذا الفصل، معظم بدائل هذه الإجرائية، بما فيها تطوير الاختبار أولاً، ولتصميم حسب الاتفاق.

1.9 تلخيص لخطوات بناء الصفوف والإجرائيات

يمكن الاقتراب من عملية بناء الصف من عدّة اتجاهات، ولكن عادةً هي عملية التصميم العام للصف، وسرد الإجرائيات الخاصة داخل الصف، وبناء إجرائيات خاصة داخل الصف، وعملية فحص بناء الصف بشكل كامل. كما يوضح الشكل 9-1، من الممكن أن تكون عملية إنشاء الصف عملية فوضوية، وذلك لأسباب كون التصميم نفسه عملية فوضوية (الأسباب الموصوفة في القسم 5-1 "تحديات التصميم").



الشكل 9-1 تختلف تفاصيل بناء صف، ولكن تحدث الإجرائيات العامة بالترتيب الظاهر هنا.

خطوات انشاء صف

إنّ الخطوات الأساسية في بناء صف هي:

أنشئ تصميم عام للصف. تشمل عملية تصميم صف العديد من القضايا المحدّدة. تحديد المسؤوليات المحدّدة للصف، وتحديد أيّة "أسرار"، سيقوم الصف بإخفائها، وتحديد بالضبط أية تجريدية ستقوم واجهة الصف بتمثيلها. تحديد فيما إذا كان سيتم اشتقاق الصف من صف أخر، أو فيما إذا كان سيسمح للصفوف الأخرى بأن تكون مشتقة من هذا الصف. تحديد العمليات العامة الرئيسية للصف، وتحديد وتصميم أية عناصر بيانات بديهية

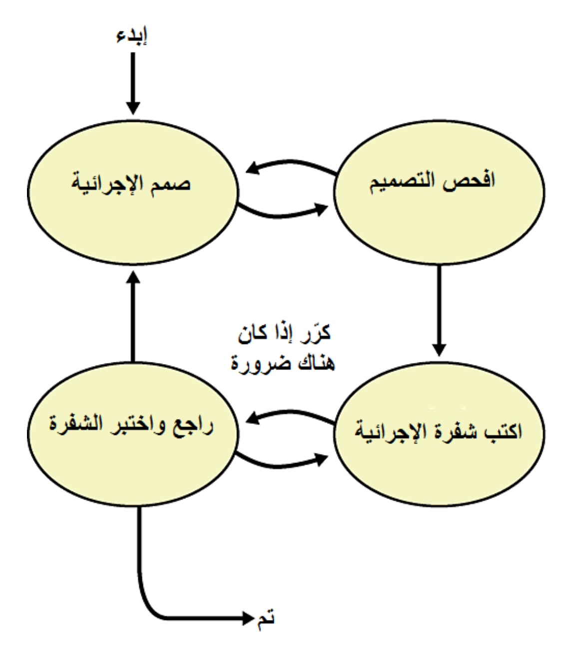
مُستخدمة داخل الصف. وتكرار هذه الخطوات عدّة مرات حسب الحاجة لإنشاء تصميم واضح للإجرائية. تمت مناقشة هذه الاعتبارات والعديد من القضايا الأخرى بالتفصيل في الفصل 6 "الصفوف الناجحة".

ابنِ كل إجرائية داخل الصف. حالما تتم عملية تعريف الإجرائيات الرئيسية في الخطوة الأولى، عليك أن تبني كل إجرائية محددة. تكشف عملية بناء كل إجرائية بشكل نموذجي عن الحاجة إلى إجرائيات إضافية، سواء أكانت ثانوية أو رئيسية، وغالباً ما تؤثر القضايا الظاهرة عند إنشاء هذه الإجرائيات الإضافية على التصميم الكلّي للصف.

راجع واختبر الصف ككل. بشكل طبيعي، يتم اختبار كل إجرائية حالما يتم توليدها. بعد أن يصبح الصف بأكمله جاهز للعمل. يجب أن يتم مراجعة واختبار الصف بأكمله من أجل القضايا التي لا يمكن اختبارها على المستوى الفردي لكل إجرائية.

خطوات بناء إجرائية

ستكون العديد من إجرائيات للصف سهلة وبسيطة التنفيذ: إجرائيات الوصول، إجرائيات التمرير إلى إجرائيات كائن أخر، وما شابه ذلك. ولكن ستكون الإجرائيات الاخرى أكثر تعقيداً، وستستفيد عملية تشكيل تلك الإجرائيات من منهج متناسق. إن الأنشطة الرئيسية المضمنة في عملية إنشاء إجرائية – تصميم الإجرائية، تفحص التصميم، كتابة شفرة الإجرائية، تفحص الشفرة – يتم تنفيذها بشكل نموذجي وفق الترتيب الظاهر في الشكل 9-2.



الشكل 9-2 يوجد نشاطات اساسية، تدخل في عملية بناء إجرائية. وعادة ما يتم تنفيذها بالترتيب المعروض هنا.

لقد طور الخبراء العديد من الطرق لتوليد الإجرائيات، وطريقتي المفضلة هي عملية برمجة الشفرة الزائفة، الموصوفة في القسم التالي.

2.9 الشفرة الزائفة من أجل الإيجابيات

يُشير مصطلح "الشفرة الزائفة" إلى كلمة غير رسمية في اللغة الإنكليزية، لوصف كيفية عمل خوارزمية أو إجرائية أو صف أو برنامج. تُحدّد عملية برمجة الشفرة الزائفة طريقة محدّدة لاستخدام الشفرة الزائفة من أجل تبسيط توليد الشفرة داخل الإجرائيات.

لأن الشفرة الزائفة تشبه اللغة الإنكليزية، فمن الطبيعي أن نفترض، بأن أية وصف بلغة تشبه اللغة الإنكليزية يجمع كل أفكارك، سيكون له نفس التأثير كأي لغة أخرى. في الممارسات العملية، ستجد أن بعض أساليب الشفرة الزائفة مُفيدة أكثر من غيرها. هُنا سنعرض الإرشادات التوجيهية للاستخدام الفعّال للشفرة الزائفة:

- استخدم عبارات باللغة الإنكليزية، تصف بدقّة عمليات محدّدة.
- تجنّب العناصر النحويّة من اللغة البرمجيّة المستهدفة. حيث تسمح الشفرة الزائفة بالتصميم على مستوى أعلى بقليل من الشفرة بحد ذاتها. عندما تقوم باستخدام بُنى لغة البرمجة، فأنت تنزل إلى مستوى أخفض، مما يلغي الفائدة الرئيسية للتصميم على مستوى أعلى، وعندها ستُتعب نفسك بالقيود النحوية للغة البرمجة غير الضرورية.
- اكتب الشفرة الزائفة على مستوى الغاية[1]. قُم بوصف معنى النهج، بدلاً من وصف كيفية تنفيذه في لغة البرمجة المستهدفة.
- اكتب الشفرة الزائفة على مستوى منخفض بما فيه الكفاية، حتى تكون عملية توليد الشفرة منها تتم تقريباً بشكل أتوماتيكي. إذا كانت الشفرة الزائفة على مستوى عالي جداً، من الممكن أن يؤدي هذا إلى تفاصيل إشكالية في الشفرة. نقّح الشفرة الزائفة في تفاصيل أكثر وأكثر، حتى يبدو أن كتابة الشفرة ستكون أسهل وأبسط.

حالما تُكتب الشفرة الزائفة، يمكنك أن تبني الشفرة حولها، وتتحول الشفرة الزائفة إلى تعليقات للغة البرمجية. يقضي هذا على كل الجهد المطلوب لكتابة التعليقات. إذا اتبعت الإرشادات التوجيهية عند كتابة الشفرة الزائفة، فستكون التعليقات كاملة وتعبيريّة. سنعرض هنا مثال عن التصميم باستخدام الشفرة الزائفة، تنتهك جميع المبادئ التي ذكرناها للتو:




```
Example of Bad Pseudocode
increment resource number by 1
allocate a dlg struct using malloc
```

[1] **إشارة مرجعية** لمزيد من التفاصيل حول التعليقات على مستوى الغاية، انظر "أنواع التعليقات" في القسم 4-32

```
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

```
مثال عن شفرة زائفة سيئة
زيادة رقم المورد بمقدار 1
خصص struct dlg باستخدام الإجرائية malloc
إذا كانت الإجرائية malloc() تُعيد NULL فأرجع العدد 1
استدعي المتغير OSrsrc_init لتهيئة مورد لنظام التشغيل
hRsrcPtr = resource number
return 0
```

ماهي الغاية من الشفرة الزائفة هذه؟ بما أنها مكتوبة بشكل ضعيف، فمن الصعب معرفة الغاية. هذه المسماة شفرة زائفة سيئة لأنها تتضمن تفاصيل كتابة الشفرة بلغة البرمجة المستهدفة، مثل hRsrcPtr* (في لغة البرمجة سي هو عبارة عن ترميز المؤشر pointer)، و malloc() (تابع محدد في لغة البرمجة سي). تُركز الشفرة الزائفة هذه على الكيفية التي سيتم فيها كتابة الشفرة، بدلاً من التركيز على معنى التصميم. إنها تدخل في تفاصيل كتابة الشفرة – فيما إذا كانت الشفرة تُعيد 0 أو 1. إذا كنت تفكّر في هذه الشفرة الزائفة من وجهة نظر كونها ستكون تعليقات جيدة أو لا للشفرة البرمجية، فستفهم أن هذا لا يساعد كثيراً.

نعرض هنا التصميم لنفس العملية ولكن باستخدام شفرة زائفة محسنّة كثيراً:

```
Example of Good Pseudocode
Keep track of current number of resources in use
If another resource is available
 Allocate a dialog box structure
 If a dialog box structure could be allocated
 Note that one more resource is in use
 Initialize the resource
 Store the resource number at the location provided by the caller
 Endif
Endif
Return true if a new resource was created; else return false
```

```
مثال عن شفرة زائفة جيدة
```

```
تتبع العدد الحالي للموارد قيد الاستخدام
إذا كان مورد آخر متاح
  خصص بنية مربع حوار
  إذا كان هناك إمكانية تخصيص بنية مربع حوار
  لاحظ وجود مورد واحد أخر قيد الاستخدام
  هيأ المورد
  خزّن رقم المورد في المكان الذي يوفره المُستدعي
  نهاية عبارة إذا (if)
نهاية عبارة إذا (if)
أعد القيمة true إذا تم إنشاء مورد جديد؛ وإلا أعد false
```

هذه الشفرة الزائفة أفضل من الأولى، لأنها مكتوبة بشكل كامل باللغة الإنكليزية؛ ولا تستخدم أية عناصر نحويّة من لغة البرمجة المستهدفة. في المثال الأول، من الممكن تنفيذ الشفرة الزائفة فقط باستخدام لغة البرمجة سي. في المثال الثاني، لا تضع الشفرة الزائفة قيود على لغة البرمجة المراد استخدامها. الشفرة الزائفة الثانية مكتوبة أيضاً على مستوى الغاية. ماذا تعني الكتلة الثانية من الشفرة الزائفة؟ من الممكن أن يكون أسهل بالنسبة لك فهم ما تعنيه أكثر من الكتلة الأولى.

على الرغم من أن الكتلة الثانية من الشفرة الزائفة مكتوبة بلغة إنكليزية غير واضحة تمامًا، إلا أنها أكثر دقة ومفصلّة بشكل كافي لتكون أساس لكتابة الشفرة بلغة برمجة ما. عندما يتم تحويل عبارات الشفرة الزائفة إلى تعليقات، فستكون عبارة عن شرح جيد لغاية الشفرة البرمجية.

هنا سنعر ض الفوائد التي يمكن تحقيقها من استخدام هذا الأسلوب من الشفرة الزائفة:

- تجعل الشـفرة الزائفة المراجعات أسـهل. حيث يمكنك مراجعة تفاصـيل التصـميم بدون تفحص شـفرة المصـدر. تجعل الشـفرة الزائفة مراجعات التصـميم منخفض المسـتوى أسـهل، وتقلّل من الحاجة إلى مراجعة الشفرة بحد ذاتها.
- تدعم الشفرة الزائفة فكرة الصقل التكراري. تبدأ أنت بتصميم عالي المستوى، ثم تصقل هذا التصميم إلى شفرة زائفة، ومن ثم تصقل هذه الشفرة الزائفة إلى شفرة المصدر. يسمح لك هذا الصقل الناجح في خطوات بسيطة بتفحص تصميمك في طريقك إلى مستويات أخفض من التفا صيل. النتيجة أنك تكشـف الأخطاء عالية المسـتوى على المسـتوى الأعلى، والأخطاء متوسـطة المسـتوى على المسـتوى الأوسـط، والأخطاء منخفضـة المسـتوى على المسـتوى الأخفض – قبل أن يصـبح أي منهم مشـكلة أو يفسد العمل على مستويات أكثر تفصيلاً.

- تجعل الشفرة الزائفة التغييرات أسهل[1]. حيث أنّ تغيير بضعة أسطر من الشفرة الزائفة أسهل من تغيير صفحة كاملة من الشفرة. هل تفضّل تغيير سطر في مخطط، أو تمزيق الجدار وإفساد الأظافر في مكانين أو أربعة أخرين؟ الأثار ليست فيزيائية في البرمجيات، ولكن مبدأ تغيير المنتج عندما يكون أكثر مرونة هو نفسه. المبادئ الأساسية لنجاح مشروع هي باكتشاف الأخطاء على "المرحلة الأقل قيمة"، وهي المرحلة التي استثمر فيها أقل جهد. يتم استثمار جهد أقل بكثير في مرحلة الشفرة الزائفة من مراحل التشفير الكامل والاختبار وتصحيح الأخطاء، لذلك من الأوفر اكتشاف الأخطاء باكراً.
- تقلّل الشفرة الزائفة من الجهد المطلوب لكتابة التعليقات. في سيناريو كتابة الشفرة النموذجي، تقوم أولاً بكتابة الشفرة، ومن ثم تُضيف التعليقات. ولكن في عملية برمجة الشفرة الزائفة، تُصبح عبارات الشفرة الزائفة هي التعليقات، لذلك في الواقع يأخذ إزالة التعليقات المزيد من العمل من تركها.

- من الأسهل المحافظة على الشفرة الزائفة من المحافظة على الصيغ الاخرى لتوثيق التصميم. باستخدام منهجيات أخرى، يتم فصل التصميم عن كتابة الشفرة، وعندما تحدث تغييرات، يفشل الاثنين في الاتفاق. ولكن باستخدام عملية برمجة الشفرة الزائفة، تُصبح عبارات الشفرة الزائفة تعليقات في الشفرة. وطالما أنه يتم الحفاظ على التعليقات المضمنة، فإن توثيق الشفرة الزائفة للتصميم تكون دقيقة.

من الصعب أن تتفوق الشفرة الزائفة كأداة للتصميم المفصّل. حيث يُفضل المبرمجين الشفرة الزائفة، للطريقة التي تسهّل فيها عملية البناء في لغة البرمجة، وعلى قدرتها على مساعدتهم في الكشف عن تصاميم غير مفصلّة بشكل كافي، ومن أجل توفيرها سهولة التوثيق وسهولة التعديل (رامسي، أتوود، وفان دورين 1983) (Ramsey, Atwood, and Van Doren 1983). إن الشفرة الزائفة ليست الأداة الوحيدة للتصميم المُفصّل، ولكن تُعد الشفرة الزائفة وعملية برمجة الشفرة الزائفة أدوات مفيدة في صندوق أدوات المبرمج الخاص بك.

3.9 بناء الإجرائيات باستخدام عملية برمجة الشفرة الزائفة

يصف هذا القسم النشاطات المتضمنة في عملية بناء الإجرائية، وهي هذه:

- تصميم الإجرائية
- كتابة شفرة الإجرائية

[1] **اقرأ أيضا** لمزيد من المعلومات حول فوائد جعل التغييرات في المرحلة الاقل قيمة، انظر أندي غروف في إدارة الخرج العالي (غروف 1983) - Andy Grove's High Output Management (Grove 1983)

- تفحص الشفرة
- إزالة الأخطاء المغفلة
- التكرار حسب الحاجة

تصميم الإجرائية

بعد قيامك بتعريف إجرائيات الصف، فإن الخطوة الأولى لبناء أيّة من الإجرائيات المعقدة للصف هي بتصميمها.[1] افترض على سبيل المثال، أنك ترغب بكتابة إجرائية، تقوم بإخراج رسالة خطأ حول أخطاء في الشفرة، وافترض أنك تستدعي الإجرائية ()ReportErrorMessag. وهذه مواصفات غير رسمية لـ ()ReportErrorMessag:

> تأخذ الإجرائية ()ReportErrorMessage شفرة الخطأ كمعامل دخل، ومن ثم تقوم بإخراج رسالة الخطأ المرتبطة بالشفرة. إن هذه الإجرائية مسؤولة عن التعامل مع الشفرات غير الصالحة. إذا كان البرنامج يعمل بشكل تفاعلي، فإن الإجرائية ()ReportErrorMessage تُظهر رسالة الخطأ للمستخدم. أما إذا كان البرنامج يعمل في نمط سطر الأوامر، فإن الإجرائية ()ReportErrorMessage تقوم بتسجيل رسالة الخطأ إلى ملف الرسائل. بعد إخراج رسالة الخطأ، تُعيد الإجرائية ()ReportErrorMessage, قيمة حالة، تُشير إلى نجاح أو فشل الإجرائية.

يستخدم الجزء الباقي من هذ الفصل هذه الإجرائية كمثال. ويصف الجزء الباقي من هذا الفصل كيفية تصميم الإجرائية.

تحقق من المتطلبات الأساسية[2]. قبل القيام بأي عمل على الإجرائية بحد ذاتها، تحقق فيما إذا كان عمل الإجرائية مُعرّف بشكل جيد، ويناسب التصميم الكامل. تحقق من موافقة الإجرائية المستدعية، وإن كان بمقدار ضئيل، لمتطلبات المشروع.

عرّف المشكلة التي ستقوم الإجرائية بحلّها. صّرح عن المشكلة التي ستحلها الإجرائية، بتفاصيل كافية لتوليد الإجرائية. إذا كان التصميم عالي المستوى مُفصّل بما فيه الكفاية، فإن معظم العمل قد تمّ للتو. يجب على التصميم عالي المستوى أن يحدّد على الأقل التالي:

- المعلومات التي ستخفيها الإجرائية

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول الجوانب الأخرى من التصميم، انظر الفصول من 5 إلى 8.

[2] **إشارة مرجعية** لمزيد من التفاصيل حول تفحص المتطلبات الأساسية، انظر الفصل الثالث "قِس مرتين، اقطع مرة: المتطلبات الاوليّة"، وانظر الفصل الرابع " قرارات البناء الرئيسية ".

- مدخلات الإجرائية
- المخرجات من الإجرائية
- الشـروط المسـبقة المضـمون صـحتها قبل اسـتدعاء الإجرائية[1] (قيم الدخل داخل مجالات معينة، والتدفقات streams المعرّفة، والملفات مفتوحة أو مغلقة، والمخازن المؤقتة مملوءة أو ممسوحة،...).
- الشـروط اللاحقة التي تضـمنها الإجرائية أن تكون صـحيحة بعد تمرير التحكم رجوعاً إلى المُسـتدعي (قيم الخرج داخل مجالات محددة، والتدفقات المعرّفة، والملفات مفتوحة أو مغلقة، والمخازن المؤقتة مملوءة أو ممسوحة،..).

هنا سنعرض كيف يتم التعامل مع هذه الاهتمامات في المثال ()ReportErrorMessage:

- تُخفي الإجرائية معاملين: نص رسالة الخطأ، وطريقة المعالجة الحالية (تفاعلية، أو سطر الأوامر).
- لا يوجد أية شروط مسبقة مضمونة للإجرائية.
- دخل الإجرائية هو عبارة عن خطأ في الشفرة.
- يتم اسـتدعاء نوعين من الخرج: الاول هو عبارة عن رسـالة الخطأ، والثاني هو الحالة التي تّعيدها الإجرائية ()ReportErrorMessage للإجرائية المُستدعيّة.
- تضمن الإجرائية أن قيمة الحالة ستأخذ إما قيمة نجاح أو فشل.

سمِّ الإجرائية.[2] من الممكن أن يكون تسمية الإجرائية أمرا تافها، ولكن أسماء الإجرائيات الجيدة هي دليل على برنامج ممتاز، وإيجاد الاسماء الجيدة ليس بالأمر السهل. بشكل عام، يجب أن تملك الإجرائية اسم واضح خالي من الغموض. إذا كانت لديك مشكلة باختيار الاسم الجيد للإجرائية، فهذا يعني أن الغرض من الإجرائية غير واضح. إن الاسم الضعيف الواهن هو كسياسي في الحملة الانتخابية. حيث يبدو كأنه يقوم بشيء ما، ولكن عندما تأخذ نظرة فاحصة، لا يمكنك معرفة ما يعنيه. إذا كان هناك إمكانيه لجعل الاسم أكثر وضوحاً، افعل ذلك. إذا كان ينتج الاسم الضعيف من التصميم الضعيف، فانتبه إلى إشارة التحذير. أي عّد للوراء، وأصلح التصميم. في المثال، إن ()ReportErrorMessage خالي من الغموض. إنه اسم جيد.

[1] **إشارة مرجعية** لمزيد من التفاصيل حول الشروط المسبقة والشروط اللاحقة، انظر "استخدم التأكيدات للتوثيق تحقق من الشروط المسبقة والشروط اللاحقة " في القسم 8-2.

[2] **إشارة مرجعية** لمزيد من التفاصيل حول تسمية الإجرائيات، انظر القسم 7-3 "الأسماء الجيدة للإجرائية"

اتخذ قرار حول كيفية اختبار الإجرائية[1]. في الوقت الذي تكتب فيه الإجرائية، فكّر في كيفية اختبارها. إن هذا مفيد لك عندما تقوم بوحدة اختبار، ومفيد للمختبر، الذي يختبر الإجرائية بشكل مستقل.

في المثال، إن الدخل بسيط، لذلك تستطيع أن تخطط لاختبار الإجرائية `ReportErrorMessage()` مع كل الاخطاء المتاحة للشفرة، ومع مجموعة متنوعة من الشفرات البرمجية غير الصالحة.

ابحث عن الوظائف المتاحة في المكاتب القياسية. إن الطريقة المفردة الأفضل لتطوير كل من جودة الشفرة والإنتاجية، هو بإعادة استخدام مقاطع الشفرة الجيدة. إذا وجدت نفسك تتصارع لتصميم إجرائية، تبدو معقدة كلياً، اسال نفسك فيما إذا كانت بعض أو كل وظائف الإجرائية موجودة للتو في مكتبة الشفرة للغة البرمجة، أو في المنصّة، أو في الأدوات التي تستخدمها. اسأل نفسك فيما إذا كانت الشفرة متاحة في مكتبة للنصوص البرمجية، مخزنة في جهاز الحاسوب. لقد تم مسبقاً اختراع العديد من الخوارزميات، وتم اختبارها، ومناقشتها في النصوص المرجعية، وتمت مراجعتها، وتطويرها. بدلاً من صرف وقتك على اختراع شيء ما، عندما يوجد شخص قد كتب مسبقاً رسالة دكتوراه عنها، فقط اصرف بضعة دقائق على مراجعة النص البرمجي المكتوب مسبقاً، وتأكد من عدم قيامك بمزيد من العمل أكثر ممّا هو ضروري.

فكّر حول معالجة الاخطاء. فكّر بكل الأشياء التي من الممكن أن تُصبح خاطئة في الإجرائية. فكّر بقيم الدخل السيئة، والقيم غير الصالحة المُعادة من الإجرائيات الاخرى، وما إلى ذلك.

يمكن للإجرائيات أن تعالج الأخطاء بالعديد من الطرق، وعليك أن تختار بوعي كيفية معالجة الأخطاء. إذا كانت هيكلة البرنامج تُعرّف استراتيجية معالجة أخطاء البرنامج، فيمكنك بسهولة تتبع هذه الاستراتيجية. وإلا، فعليك أن تقرّر أيّة طريقة ستعمل أفضل مع إجرائية محدّدة.

فكّر في الكفاءة. حسب الحالة التي لديك، يمكنك معالجة الكفاءة بواحدة من طريقتين. في الحالة الأولى، في الغالبية العظمى من الأنظمة، الكفاءة ليست بالموضوع الحاسم. في حالة كهذه، تأكّد فيما إذا كانت واجهة الإجرائية مُجرّدة بشكل جيد، وفيما إذا كان النص البرمجي لها قابلاً للقراءة بحيث يمكنك أن تحسّنه فيما بعد إذا احتجت إلى ذلك. إذا كان لديك تغليف جيد، فيمكنك استبدال تنفيذ اللغة عالية المستوى البطيئة بخوارزمية أفضل أو بتنفيذ منخفض المستوى سريع، ولن تؤثر على الإجرائيات الأخرى.

في الحالة الثانية، في أنظمة قليلة الأداء، الكفاءة هو موضوع حاسم[2]. من الممكن أن يتعلق موضوع الأداء باتصالات قاعدة البيانات الضعيفة، أو بالذاكرة المحدودة، أو بعدد قليل من مقابض التحكم المتاحة، أو بقيود

[1] اقرأ أيضا من أجل طريقة أخرى للبناء، التي تُركز على كتابة حالات الاختبار أولاً، انظر "التطوير بالاختبار: على سبيل المثال" بيك 2003 (Beck 2003)

[2] **إشارة مرجعية** لمزيد من التفاصيل حول الكفاءة، انظر الفصل 25 "استراتيجيات ضبط الشفرة"، والفصل 26 "تقنيات ضبط الشفرة".

التوقيت، أو بضعف بعض المصادر الأخرى. يجب أن تُشير الهيكلة إلى عدد المصادر المسموح استخدامها لكل إجرائية (أو صف)، ومدى السرعة التي يجب أن تنفذ فيها عمليات الإجرائية.

صمّم الإجرائية بحيث توافق مصادرها وأهداف السرعة لها. إذا كان يبدو أن المصادر أو السرعة هي الموضوع الأكثر أهميةً، صمّم بطريقة تقوم بها بمقايضة المصادر بالسرعة أو العكس بالعكس. من المقبول أثناء عملية البناء الاوليّة للإجرائية، أن يتم ضبط الإجرائية بحيث تُوافق إمكانيات السرعة والمصادر المتاحة.

إلى جانب الطريقتين المُقترحتين لهاتين الحالتين العامتين، إنه بالعادة هدر للجهد العمل على الكفاءة على مستوى الإجرائيات المنفردة. تأتي الأمثلّة الكبرى من عملية تنقية التصميم عالي المستوى، وليس من الإجرائيات المنفردة. أنت بالعادة تستخدم أمثلّة صغيرة فقط عندما لا يدعم التصميم عالي المستوى أهداف أداء النظام، ولن تتمكن من معرفة هذا إلا عندما يتم إنجاز كامل البرنامج. لا تهدر الوقت على تحسينات إضافية حتى تدرك أنه يوجد حاجة لهذه التحسينات.

ابحث عن الخوارزميات وأنواع البيانات. إذا لم تكن الوظيفة متاحة في المكتبات، من الممكن أن تكون مشروحة في كتاب للخوارزميات. قبل أن تبدأ بكتابة شفرة معقدة من الصفر، تفحص كتاب للخوارزميات، لترى فيما إذا كانت الشفرة مُتاحة مسبقاً. إذا كنت تستخدم خوارزمية معرّفة مُسبقاً، تأكد من تعديلها بشكل صحيح لتوافق لغة البرمجة التي تستخدمها.

اكتب الشفرة الزائفة. قد لا يكون لديك الكثير من الكتابة بعد الانتهاء من الخطوات السابقة. الغرض الأساسي من الخطوات هو تأسيس توجه عقلي مُفيد عندما تقوم بكتابة إجرائية.

عندما تكون الخطوات الأوليّة كاملة، عندها تستطيع أن تبدأ بكتابة الإجرائية كشفرة زائفة عالية المستوى[1]. تحرّك قدماً واستخدم محرّر البرمجة الخاص بك أو البيئة المتكاملة لكتابة الشفرة الزائفة – سيتم استخدام الشفرة الزائفة قريباً كأساس لكتابة الشفرة بلغة البرمجة.

ابدأ من الشيء العام، ومن ثم اعمل باتجاه شيء ما أكثر تخصصاً. الجزء الاكثر عامةً من الإجرائية هو التعليق الرأسي، الذي يصف ما هو مفترض أن تقوم به الإجرائية، لذلك بدايةً اكتب عبارة مختصرة عن الغاية من الإجرائية. سيساعدك كتابة هذه العبارة على الفهم الواضح للإجرائية. المشكلة أثناء كتابة التعليق العام هو عبارة عن تحذير أنه عليك أن تفهم دور الإجرائية في البرنامج بشكل أفضل. بشكل عام، من الصعب تلخيص هدف الإجرائية، يجب أن تفترض أنه على الأرجح هناك شيء ما خطأ. هنا نعرض مثال عن تعليق رأسي يصف إجرائية:

1 **إشارة مرجعية** تفترض هذه المناقشة استخدام تقنيات التصميم الجيدة لتوليد نسخة شفرة زائفة من الإجرائية. لمزيد من التفاصيل حول التصميم، انظر الفصل 5 "التصميم في البناء"

```
Example of a Header Comment for a Routine
This routine outputs an error message based on an error code supplied
by the calling routine. The way it outputs the message depends on the
current processing state, which it retrieves on its own. It returns a
value indicating success or failure.
```

مثال عن تعليق رأسي لإجرائية

تُخرج هذه الإجرائية رسالة خطأ استناداً إلى الخطأ في الشفرة الذي تم توفيره من الإجرائية الداعية. تعتمد الطريقة التي تُخرج فيها الرسالة على حالة المعالجة الحالية، التي تُسترجع من تلقاء نفسها. تقوم بإرجاع قيمة تشير إلى النجاح أو الفشل.

بعد كتابتك للتعليق العام، اكتب الشفرة الزائفة للإجرائية. هنا نعرض مثال عن شفرة زائفة لهذا المثال:

```
Example of Pseudocode for a Routine
This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
set the default status to "fail"
look up the message based on the error code
if the error code is valid
 if doing interactive processing, display the error message
 interactively and declare success
 if doing command line processing, log the error message to the
 command line and declare success
if the error code isn't valid, notify the user that an internal error
has been detected
return status information
```

مثال عن شفرة زائفة لإجرائية

تُخرج هذه الإجرائية رسالة خطأ استناداً إلى الخطأ في الشفرة الذي تم توفيره من الإجرائية الداعية. تعتمد الطريقة التي تُخرج فيها الرسالة على حالة المعالجة الحالية، التي تُسترجع من تلقاء نفسها. تقوم بإرجاع قيمة تشير إلى النجاح أو الفشل.

ضبط الحالة الافتراضية على "فشل"

البحث عن الرسالة بناء على شفرة الخطأ

إذا كانت شفرة الخطأ صالحة

إذا قمت بإجراء معالجة تفاعلية، اعرض رسالة الخطأ على نحو تفاعلي وأعلن النجاح

إذا قمت بإجراء معالجة سطر الأوامر، قم بتسجيل رسالة الخطأ إلى سطر الأوامر وأعلن النجاح
إذا كانت شفرة الخطأ غير صالحة، فأخبر المستخدم بأنه تم اكتشاف خطأ داخلي
أعد معلومات الحالة

لاحظ مرة ثانية أن الشفرة الزائفة مكتوبة على مستوى عالي. من المؤكد أنها غير مكتوبة باستخدام لغة برمجة. بدلاً من ذلك، إنها تعبّر باستخدام اللغة الإنكليزية عن الهدف من الإجرائية.

فكّر في البيانات.[1] يمكنك تصميم بيانات الإجرائية في نقاط مختلفة عديدة من عملية بناء الإجرائية. في هذا المثال، البيانات بسيطة، ومعالجة البيانات ليست بجزء بارز من الإجرائية. أما إذا كانت معالجة البيانات جزء بارز من الإجرائية، فإنه يستحق أن تفكّر بالقطع الأساسية للبيانات قبل التفكير بمنطق الإجرائية. إن تعريف أنواع البيانات الأساسية أمر مفيد وجوده عندما تقوم بتصميم المنطق للإجرائية.

تفحص الشفرة الزائفة[2]. عند قيامك بكتابة الشفرة الزائفة وتصميم البيانات، قُم بمراجعة الشفرة الزائفة التي قمت بكتابتها. انظر إلى الشفرة من بعيد وفكّر في الطريقة التي ستقوم بشرحها إلى شخص أخر.

اسأل شخص أخر لتفحص الشفرة الزائفة التي كتبتها أو ليستمع إلى شرحك لها. من الممكن أن تفكر أنه لأمر سخيف أن تتطلب من شخص ما قراءة 11 سطر من الشفرة الزائفة، ولكنك ستتفاجأ. حيث تستطيع الشفرة الزائفة أن تجعل افتراضاتك والأخطاء عالية المستوى أكثر وضوحاً، مما تستطيع أن تفعله الشفرة المكتوبة بلغة برمجة ما. وأيضاً الناس أكثر استعداداً على مراجعة بضعة خطوط من الشفرة الزائفة، من مراجعة 35 سطر من سي++ أو جافا.

تأكّد من أنك تملك فهم مريح وسهل لما تفعله الإجرائية وكيف تفعل هذا. إذا كنت لا تفهم هذا نظريّاً على مستوى الشفرة الزائفة، فما هي فرصة فهمك له على مستوى لغة البرمجة؟ وإذا كنت لا تفهمه، فمن سيفهمه غيرك؟

جرّب عدة أفكار في الشفرة الزائفة، واترك الخيار الافضل (كرّر)[3]. جرب بقدر ما تستطيع العديد من الافكار في الشفرة الزائفة، قبل أن تبدأ بعملية كتابة النص البرمجي. بمجرد بداية كتابة النص البرمجي، سيكون لديك ترابط عاطفي مع النص البرمجي، وسيصبح من الصعب ارتكاب تصميم سيء. إن الفكرة العامة هي بتكرار الإجرائية في الشفرة الزائفة، حتى تصبح الشفرة الزائفة بسيطة بما فيه الكفاية لتتمكن من إضافة النص البرمجي لكل عبارة، وترك الشفرة الزائفة الأصلية كتوثيق للنص البرمجي. قد تبدو بعض الشفرات الزائفة المكتوبة من المحاولة الاولى، على مستوى عالي كافي لإمكانية تفكيكها. تأكد من تفكيكها. إذا لم تكن متأكد من

1 **إشارة مرجعية** لمزيد من التفاصيل حول الاستخدام الفعال للمتغيرات، انظر الفصول من 10 إلى 13.

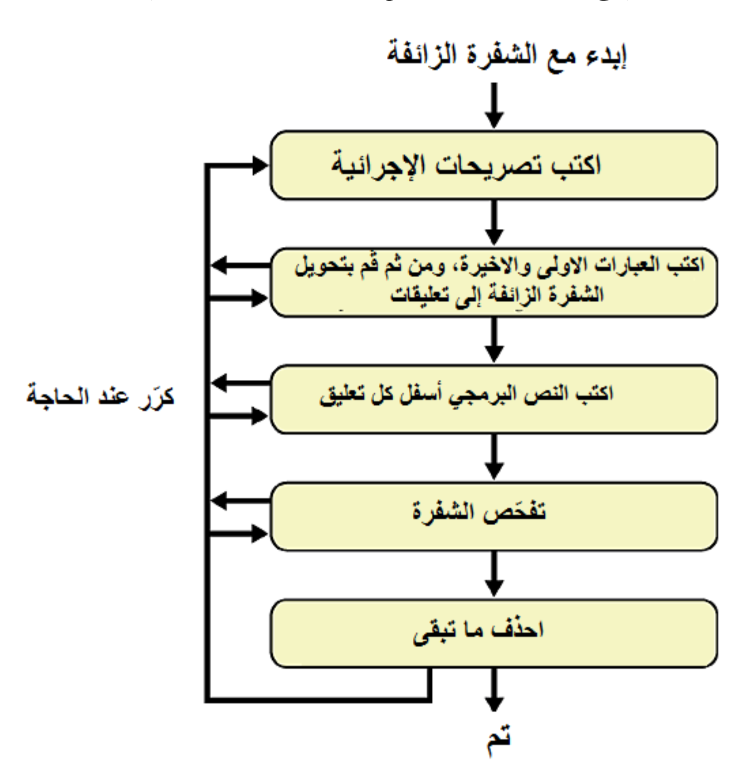
2 **إشارة مرجعية** لمزيد من التفاصيل حول تقنيات المراجعة، انظر الفصل 21 " البناء التعاوني ".

3 **إشارة مرجعية** للمزيد حول التكرار، انظر القسم 34-8 "كرر مراراً وتكراراً"

كيفية كتابة النص البرمجي لشيء ما، فاستمر بالعمل مع الشفرة الزائفة، حتى تُصبح متأكد. استمر بتنقيح وتفكيك الشفرة الزائفة، حتى تصل إلى مرحلة يبدو فيها الاستمرار بالعمل مع الشفرة الزائفة هدراً للوقت بدلاً من البدء بكتابة النص البرمجي الفعلي.

كتابة شفرة الإجرائية

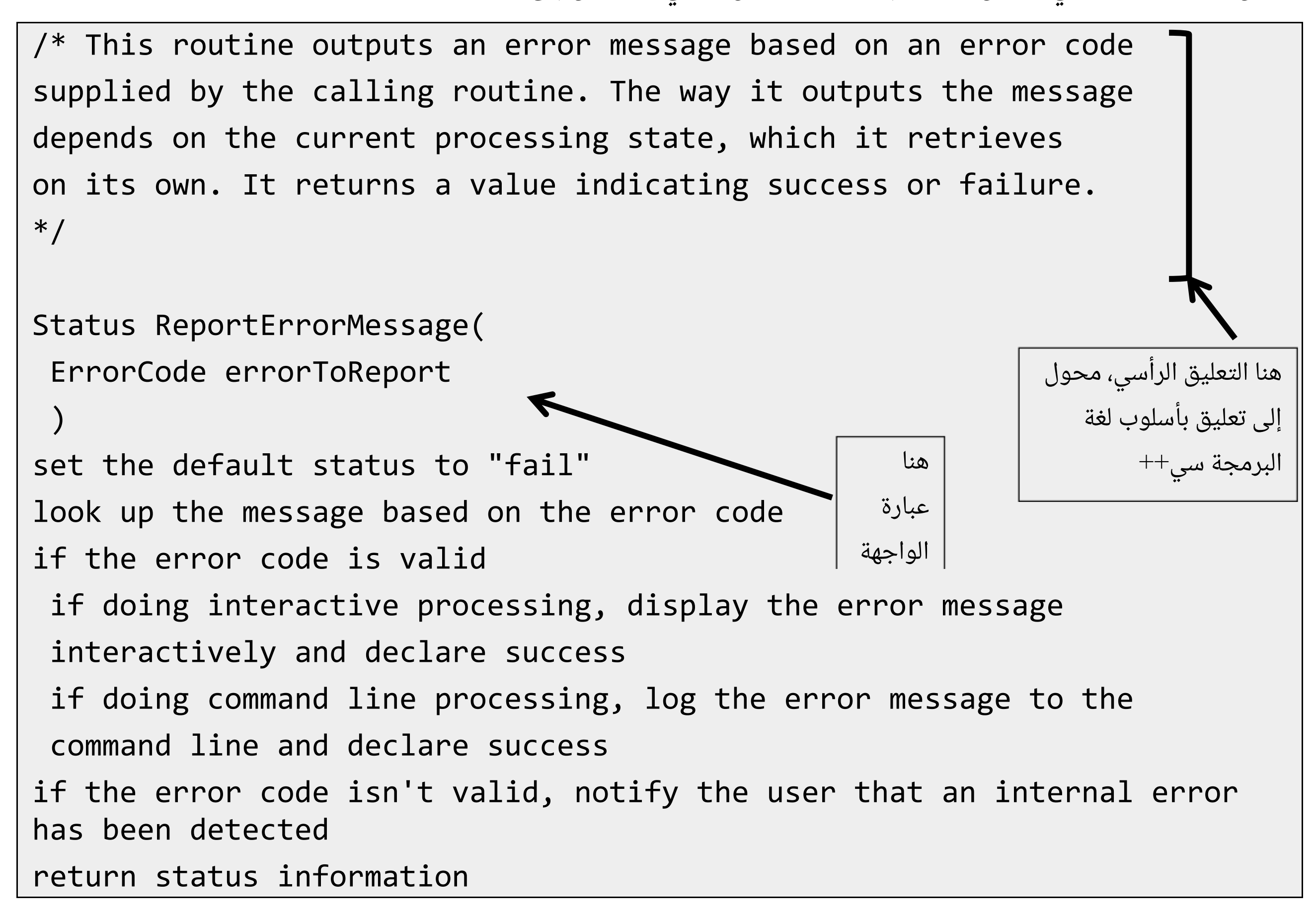
بمجرد تصميمك للإجرائية، ابدأ بناءها. يمكن أداء خطوات البناء بترتيب قياسي ضيق، ولكن اشعر بالحرية في إمكانية تغيير الخطوات إذا احتجت إلى ذلك. يُظهر الشكل 9-3 خطوات بناء إجرائية.



الشكل 9-3 ستقوم بتنفيذ كل هذه الخطوات عند تصميم إجرائية، ولكن ليس بالضرورة بترتيب معين.

اكتب تصريحات الإجرائية. اكتب عبارة الواجهة للإجرائية – تصريح التابع في لغة البرمجة سي++، تصريح المنهج (method) في لغة البرمجة جافا، تصريح التابع أو الإجرائية الفرعية في لغة البرمجة "مايكروسوفت فيجوال بيسيك"، أو ما تدعوه لغة البرمجة التي تستخدمها. قم بتحويل التعليق الرأسي الأصلي إلى تعليق في لغة البرمجة. اتركه في موقع أعلى الشفرة الزائفة التي كتبتها للتو. هنا مثال عن عبارات الواجهة والعبارة الرأسية لإجرائية في لغة البرمجة سي++:

مثال بلغة البرمجة سي++ عن واجهة إجرائية، وتعليق رأسي مُضافين إلى شفرة زائفة.




```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
 ErrorCode errorToReport
 )
set the default status to "fail"
look up the message based on the error code
if the error code is valid
 if doing interactive processing, display the error message
 interactively and declare success
 if doing command line processing, log the error message to the
 command line and declare success
if the error code isn't valid, notify the user that an internal error
has been detected
return status information
```

هذا هو الوقت المناسب لتقديم ملاحظات حول افتراضات أية واجهة. في هذه الحالة، يجب على متغير الحالة errorToReport أن يكون مباشر ومكتوب لغرض محدد، لذلك لن تكون هناك حاجة إلى توثيقه.

قُم بتحويل الشفرة الزائفة إلى تعليقات عالية المستوى. اكتب أول و آخر عبارة في لغة البرمجة سي++: { }. ثم قُم بتحويل الشفرة الزائفة إلى تعليقات. هنا سنعرض كيف سيبدو هذا في المثال:

مثال بلغة البرمجة سي++ عن كتابة العبارات الأولى ولأخيرة حول الشفرة الزائفة.

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/
Status ReportErrorMessage(
 ErrorCode errorToReport
 ) {
// set the default status to "fail"
 // look up the message based on the error code
```

من هنا تم تحويل
عبارات الشفرة الزائفة
إلى تعليقات في لغة
البرمجة سي++

```
 // if the error code is valid
 // if doing interactive processing, display the error message
 // interactively and declare success
 // if doing command line processing, log the error message to the
 // command line and declare success
 // if the error code isn't valid, notify the user that an
 // internal error has been detected
 // return status information
}
```

عند هذه النقطة من التصميم، طبيعة الإجرائية أصبحت واضحة. عملية التصميم كاملة، ويمكنك أن تفهم الألية التي تعمل بها الإجرائية حتى بدون رؤية أية نص برمجي. يجب أن تشعر بأن عملية تحويل الشفرة الزائفة إلى نص برمجي بلغة برمجية ما، طبيعية وسهلة وتتم بشكل ميكانيكي. إذا لم تشعر بهذا، استمر بعملية التصميم في الشفرة الزائفة، حتى يُصبح التصميم متين.

اكتب الشفرة أسفل كل تعليق.[1] اكتب النص البرمجي الخاص بكل سطر من تعليقات الشفرة الزائفة. العملية أشبه بكتابة تقرير. أولاً عليك كتابة الخطوط العريضة، ومن ثم عليك كتابة فقرة لكل نقطة من الخطوط العريضة. يصف كل تعليق الشفرة الزائفة كتلة أو فقرة من الشفرة. مثل طول الفقرات الأدبية، فإنه يتغير طول الفقرات البرمجية بالاعتماد على الأفكار التي تُعبر عنها، وتعتمد جودة الفقرات على حيوية وتركيز الأفكار فيها.

في هذا المثال، يتم تحويل أول سطرين من تعليقات الشفرة الزائفة إلى سطرين برمجيين.

مثال بلغة البرمجة سي++ عن التعبير عن تعليقات الشفرة الزائفة كشفرة

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/
Status ReportErrorMessage(
 ErrorCode errorToReport
 ) {
 // set the default status to "fail"
Status errorMessageStatus = Status_Failure;
```

هنا النص البرمجي المُضاف

1 **إشارة مرجعة**: هذه الحالة التي تعمل فيها كتابة الاستعارة بشكل جيد - في الصغير. لنقد تطبيق كتابة الاستعارة في الكبير، انظر "فن البرمجيات: كتابة الشفرة" في القسم 2. 3.

```
 // look up the message based on the error code
 Message errorMessage = LookupErrorMessage( errorToReport );
 // if the error code is valid
 // if doing interactive processing, display the error message
 // interactively and declare success
 // if doing command line processing, log the error message to the
 // command line and declare success
 // if the error code isn't valid, notify the user that an
 // internal error has been detected
 // return status information
}
```

هنا المتغير الجديد errorMessage

هذه هي بداية للشفرة. تم استخدام المتغير errorMessage، لذلك يجب التصريح عنه. إذا أردت أن تعلّق على هذا المثال، سطرين برمجيين من أجل تعليقين، سيكون هذا دائماً مبالغ فيه. في هذه الطريقة، على كل حال، المهم هو المحتوى الدلالي للتعليقات، وليس ما هو عدد السطور البرمجية التي تمثلها. التعليقات موجودة للتو، ويشرحون الغاية من الشفرة، لذلك اتركهم.

الشفرة الجاهزة تحت، حيث تم إضافة نص برمجي لكل تعليق:

مثال بلغة البرمجة سي++ عن إجرائية مولدة بالكامل باستخدام عملية برمجة الشفرة الزائفة

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/
Status ReportErrorMessage(
 ErrorCode errorToReport
 ) {
 // set the default status to "fail"
 Status errorMessageStatus = Status_Failure;
 // look up the message based on the error code
 Message errorMessage = LookupErrorMessage( errorToReport );
 // if the error code is valid
if ( errorMessage. ValidCode() ) {
 // determine the processing method
 ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();
 // if doing interactive processing, display the error message
 // interactively and declare success
 if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
```

من هنا تمت إضافة النص البرمجي المطلوب لكل تعليق.

```
DisplayInteractiveMessage( errorMessage. Text() );
errorMessageStatus = Status_Success;
}
// if doing command line processing, log the error message to the
// command line and declare success
else if ( errorProcessingMethod == ProcessingMethod_CommandLine ) {
CommandLine messageLog;
if ( messageLog. Status() == CommandLineStatus_Ok ) {
messageLog. AddToMessageQueue( errorMessage. Text() );
messageLog. FlushMessageQueue();
errorMessageStatus = Status_Success;
}
else {
// can't do anything because the routine is already error processi
}
else {
// can't do anything because the routine is already error processing
}
}
// if the error code isn't valid, notify the user that an
// internal error has been detected
else {
DisplayInteractiveMessage(
"Internal Error: Invalid error code in ReportErrorMessage()"
);
}
// return status information
return errorMessageStatus;
}
```

هذا النص البرمجي مُرشح جيد، ليتم تفكيكه فيما بعد إلى إجرائية تكرارية جديدة، بالاسم: DisplayCommandLineMessage

هذا النص البرمجي و التعليق جديدين، وهما نتيجة عدم صحة اختبار if

هذا النص البرمجي والتعليق أيضاً جديدين

تم زيادة على كل تعليق سطر او أكثر من السطور البرمجية. تُمثل كل كتلة من الشفرة فكرة متكاملة على أساس التعليق. تمت المحافظة على التعليقات، لتوفير شرح عالي المستوى للشفرة. تم التصريح وتعريف المتغيرات قريباً من المكان التي تُستخدم فيها. يجب بشكل طبيعي شرح كل تعليق باستخدام حوالي 2 إلى 10 خطوط برمجية. (لأن هذا المثال فقط لغاية التوضيح، حيث فيه النص البرمجي على أخفض مستوى من التوسيع الذي يمكن ان تراه في الحياة العملية).

الآن انظر إلى المواصفات في الصفحة 321 وعلى الشفرة الزائفة الاوليّة في الصفحة 326. تم توسيع خمس جمل من المواصفات إلى 15 سطر من الشفرة الزائفة (حسب الطريقة التي تعد فيها السطور)، وهذا الشفرة الزائفة تم تحويلها إلى إجرائية على صفحة طويلة. على الرغم من أن المواصفات كانت مفصلّة، ولكن تطلب توليد الإجرائية عمل تصميم كبير على الشفرة الزائفة والنص البرمجي. ذلك التصميم منخفض المستوى هو أحد أسباب، لماذا "التشفير" مهمه بديهية، ولماذا موضوع الكتاب مهم.

تحقق فيما إذا يجب تحليل (تفكيك) الشفرة. في بعض الحالات، سوف ترى كتلة كبيرة من النص البرمجي تحت خط واحد مبدئي من الشفرة الزائفة. في هذه الحالة، عليك ان تأخذ بعين الاعتبار واحدة من الإجرائيتين التالتين:

- ضع النص البرمجي أسفل التعليق في إجرائية جديدة[1]. إذا وجدت ان سطر واحد من الشفرة الزائفة قد ادى إلى كتالة نص برمجي أكبر من المتوقع، عندها عليك ان تضـع هذا النص البرمجي في إجرائية خاصــة به. اكتب النص البرمجي الذي يســتدعي الإجرائية، بما في ذلك اســم الإجرائية. إذا كنت قد اســتخدمت عملية برمجة الشــفرة الزائفة بشــكل جيد، فإنه يجب أن يظهر اســم الإجرائية الجديدة بسـهولة من الشـفرة الزائفة نفسـها. حالما إتمامك للإجرائية الاصـلية التي تولدها، يمكنك الغوص في الإجرائية الجديدة، وتطبيق عليها عملية برمجة الشفرة الزائفة مرة ثانية.
- طبّق عملية برمجة الشــفرة الزائفة بشــكل متكرر. بدلاً من كتابة عدة عشــرات من الســطور البرمجية أسفل سطر واحد من الشفرة الزائفة، الأفضل إعادة تفكيك السطر الأصلي من الشفرة الزائفة إلى عدة سطور إضافية في الشفرة الزائفة. وبعدها استمر بكتابة التعليمات البرمجية أسفل الخطوط الجديدة من الشفرة الزائفة.

تفحص الشفرة

بعد تصميم وتنفيذ الإجرائية، فإن الخطوة الثالثة الكبيرة من خطوات بناءها، هي تفحص فيما إذا كان الذي قمنا ببنائه صحيح. لن يتم اكتشاف أية أخطاء منسية في هذه المرحلة حتى تتم مرحلة الاختبار التالية. عندها سيكون من المكلف إيجاد هذه الأخطاء وإصلاحها، لذلك عليك أن تجد كل الأخطاء الممكنة في الإجرائية في هذه المرحلة.

قد لا تظهر مشكلة حتى يتم كتابة الشفرة للإجرائية بالكامل،[2] لعدة أسباب. قد يصبح الخطأ في الشفرة الزائفة أكثر وضوحاً في منطق التنفيذ المفصّل. قد يصبح التصميم الذي يبدو مُتقن في الشفرة الزائفة، غير بارع التنفيذ

---

[1] **إشارة مرجعية** للمزيد حول إعادة بناء التعليمات البرمجية، انظر الفصل 24 "إعادة التصنيع".

[2] **إشارة مرجعية** لمزيد من التفاصيل حول تفحص الاخطاء في الهيكلية وفي المتطلبات، انظر الفصل 3 "قس مرتين، اقطع مرة: المتطلبات الأولية".

في لغة البرمجة. من الممكن أن يكشف العمل مع التنفيذ المفصّل عن خطأ في الهيكلية، أو في التصميم العالي المستوى، أو في المتطلبات. وأخيراً قد تحوي الشفرة على خطأ بر مجي قديم الطراز – لا يوجد أحد كامل. لهذه الأسباب، يجب مراجعة النص البرمجي قبل التحرك قُدماً.

تحقق من الإجرائية عقلياً بحثاً عن الأخطاء. أول صيغة لتفحص الإجرائية هي تفحص عقلي. إن عمليتي المسح والفحص غير الرسمي المذكورتين سابقاً، هما نوعين من الفحص العقلي. الأخر هو تنفيذ كل مسار عقليا. إن تنفيذ الإجرائية عقلياً أمر صعب، وهذه الصعوبة هي أحد أسباب المحافظة على الإجرائية صغيرة قدر الإمكان. تأكد من أنك تتفحص مسارات الأسماء وقاط النهاية وكل شروط التنفيذ. قُم بهذا بنفسك، هذا ما يُدعى "فحص المكتب"، أو مع واحد او أكثر من زملائك، هذا يُدعى "مراجعة الزملاء"، أو "التفحّص " أو "التفتيش"، وذلك اعتمادا على كيفية القيام بذلك.

واحدة من الفروق الكبيرة بين المبرمجين الهواة ولمبرمجين المحترفين، هو الفرق الذي ينمو من الانتقال من الخرافة إلى الفهم. لا تُشير كلمة "خرافة" في هذا السياق إلى البرنامج الذي يخيفك، او يولد أخطاء إضافية كثيرة عند اكتمال القمر. هذا يعني استبدال المشاعر حول النص البرمجي بالفهم. إذا كنت غالباً تجد نفسك تشك بقيام المترجم البرمجي أو العتاد الصلب بخطأ، هذا يعني أنك لا تزال في مملكة الخرافات. وجدت دراسة، أُجريت منذ عدة سنوات، فقط خمسة بالمئة من الأخطاء هي أخطاء العتاد الصلب، أو المترجم البرمجي، أو أنظمة التشغيل (أوستراند وويوكر 1984) (سOstrand and Weyuker 1984). اليوم هذه النسبة على الأرجح انخفضت. يشك المبرمجين الذين انتقلوا إلى مملكة الفهم أولاً بعملهم، وذلك لأنهم يعرفون أنهم يسببون 95 بالمئة من الأخطاء. افهم دور كل سطر في الشفرة، ولماذا هناك حاجة إليها. لا يوجد أي شيء صحيح فقط لأنه يبدو أنه يعمل. إذا لم تكن تعرف لماذا يعمل، من الممكن أنه لا يعمل – فقط أنت لم تعرف هذا حتى الان.



خلاصة القول: إجرائية عاملة هذا لا يكفي. إذا لم تكن تعرف لماذا تعمل، قُم بدراستها، بمناقشتها، وبتجربتها مع تصاميم اخرى حتى تفهم لماذا تعمل.

ترجم الإجرائية بعد مراجعة الإجرائية، قُم بترجمتها. قد يبدو غير فعال الانتظار كل هذا الوقت لترجمة النص البرمجي، حيث النص البرمجي قد كان جاهز منذ عدة صفحات سابقة. باعتراف الجميع، من الممكن أن تكون قد وفرّت بعض الأعمال لو قمت بترجمة الإجرائية سابقاً، ولو سمحت لجهاز الحاسوب بالتحقق من المتغيرات غير المصرّح عنها، وتعارض بالأسماء، وإلى اخره.

سوف تستفيد بعدة طرق، على كل حال، عن طريق عدم ترجمة الإجرائية حتى وقت متأخر من العملية. السبب الرئيسي هو أنه عندما تقوم بترجمة نص برمجي جديد، فإن ساعة توقيت داخلية تبدأ بالعمل. بعد أول عملية ترجمة، سوف تزيد الضغط "سأحصل عليه صحيح فقط عن طريق عملية ترجمة ثانية". تؤدي العبارة "عملية

ترجمة ثانية" إلى تغييرات سريعة معرضة للخطأ، تأخذ المزيد من الوقت على المدى الطويل. تجنب الاندفاع إلى عملية الترجمة، عن طريق ترجمة الإجرائية فقط عندما تقتنع أنها صحيحة.

إن غاية هذا الكتاب هو إظهار كيفية تجنب عملية قرصنة بعض الأشياء مع بعض، وشغيلها لترى فيما إذا كانت تعمل. إن القيام بعملية الترجمة قبل التأكد من ان برنامجك يعمل، هي أحد أعراض عقلية القرصنة. إذا لم تكن متورطاً بعملية القرصنة والتجميع، قُم بعملية الترجمة للنص البرمجي فقط عندما تجد أنه الوقت المناسب لذلك. ولكن كن واعيا لفكر معظم الناس الذي يعتبرون "القرصنة، الترجمة، الإصلاح" طريقتهم لبرنامج عامل.

هنا نعرض بعض الإرشادات التوجيهية للحصول على أقصى استفادة من عملية ترجمة الإجرائية:

- عيّن مستوى التحذير للمترجم إلى أعلى مستوى ممكن. حيث يمكنك اكتشاف عدد مذهل من الأخطاء الصغيرة، عن طريق السماح للمترجم بالكشف عنها.
- ا ستخدم أدوات التحقق. إن المترجم الذي يفحص ما تم تنفيذه في لغة البرمجة سي، يمكن ا ستكماله باســتخدام أدوات مثل "lint". حتى النص البرمجي الذي لا تتم ترجمته، مثل HTML وجافا ســكريبت، يمكن التحقق منه باستخدام أدوات التحقق.
- اقضِ على كل أسـباب رسـائل الخطأ والتحذير. انتبه إلى ما تقوله هذه الرسـائل حول النص البرمجي الخاص بك. يُشـــير غالباً عدد كبير من التحذيرات على انخفاض جودة النص البرمجي، وعليك أن تحاول فهم كل تحذير تحصـــل عليه. في الممارســات العملية: تؤدي التحذيرات التي تراها تكراراً إلى واحدة من اثنين من الأثار المحتملة: أنت تتجاهلهم وهم يخفون وراءهم تحذيرات اخرى أكثر أهمية، أو ببســـاطة يصـــبحون مزعجين. إنه عادةً أكثر أمناً وأقل إيلاماً، إعادة كتابة النص البرمجي لحل المشكلات الأساسية، وللقضاء على التحذيرات.
-

امشِ خطوة خطوة خلال النص البرمجي في المصحح. حالما تتم ترجمة الإجرائية، ضعها داخل المصحح وامشي خطوة خطوة من خلال كل سطر في النص البرمجي. تأكّد من أن كل سطر برمجي يُنفذ ما تتوقع منه أن ينفذه. من خلال القيام بهذا سوف تكتشف العديد من الأخطاء.

اختبر النص البرمجي[1]. اختبر الشفرة من خلال استخدام حالات الاختبار، التي خططت لها عندما كنت تطور الإجرائية. قد تضطر إلى تطوير سقالة لدعم حالات الاختبار لديك – وهي النص البرمجي الذي يُستخدم لدعم الإجرائيات أثناء اختبارها، ولا يتم تضمين هذا النص البرمجي في المنتج النهائي. من الممكن ان تكون هذه

[1] **إشارة مرجعية** لمزيد من التفاصيل، انظر الفصل 22 "تطوير الاختبار"، وانظر أيضاً "بناء السقالات لاختبار الصفوف المفردة" في القسم 22-5

السقالة عبارة عن إجرائية، التي تستدعى الإجرائية الخاصة بك باستخدام بيانات الاختبار، أو يمكن أن تكون الإجرائية التي تُستدعى من قبل الإجرائية الخاصة بك.

احذف الأخطاء من الإجرائية.[1] حالما يتم كشف الخطأ، يجب حذفه على الفور. إذا كانت الإجرائية التي طورتها مشوبة بالأخطاء، فإنه يوجد احتمالات جيدة بأنها ستظل مشوبة بالأخطاء. إذا وجدت أن إجرائية مشوبة بالأخطاء بشكل غير اعتيادي، فابدأ من جديد. لا تقم بالقرصنة حول هذه الإجرائية المشوبة بالأخطاء، بل قُم بإعادة كتابتها. تُشير القرصنة عادةً إلى عدم الفهم الكامل وتضمن وجود الأخطاء الأن وفيما بعد. إن توليد تصميم جديد كلياً لإجرائية مشوبة بالأخطاء يؤتي ثماره. يوجد بعض الأشياء أكثر إرضاءً من إعادة كتابة إجرائية إشكالية وعدم إيجاد خطأ أخر فيها أبداً.

احذف ما تبقى

بمجدر انتهاءك من تفحص النص البرمجي بحثاً عن الأخطاء، تحقق منه بحثاً عن المواصفات العامة الموصوفة في هذا الكتاب. يمكنك اتخاذ مجموعة من خطوات حذف البيانات الفائضة، للتأكد من ان جودة الإجرائية تناسب معاييرك:

- تحقق من واجهة الإجرائية. تأكّد من أنه تم أخذ بالحسبان جميع بيانات الدخل وبيانات الخرج، وتم استخدام جميع الوسطاء. لمزيد من التفاصيل، انظر القسم 7-5 "كيفية استخدام وسطاء الإجرائية".

- تحقق من جودة التصميم العام. تأكد من قيام الإجرائية بشيء واحد، وبقيامها به بشكل جيد، ومن تباعدها عن الإجرائيات الأخرى، وأنها مصممة بشكل دفاعي. لمزيد من التفاصيل، انظر الفصل 7 "الإجرائيات عالية الجودة".
- تحقق من متغيرات الإجرائية. تحقق من أسماء المتغيرات غير الدقيقة، الكائنات غير المستخدمة، المتغيرات غير المصرح عنها، الكائنات التي تم تهيئتها بشكل غير صحيح، وإلى أخره. لمزيد من التفاصيل، انظر الفصول حول استخدام المتغيرات، من الفصل 10 إلى الفصل 13.
- تحقق من عبارات الإجرائية ومنطقها. تحقق من الأخطاء واحدة تلو الأخرى، الحلقات غير النهائية، الحلقات المتداخلة غير المناسبة، وتسرب الموارد. لمزيد من التفاصيل، انظر الفصول حول العبارات البرمجية من الفصل 14 إلى 19.

[1] **إشارة مرجعية** لمزيد من التفاصيل، انظر الفصل 23 " تصحيح الأخطاء"

- تحقق من تنسيق الإجرائية. تأكد من استخدام مساحة بيضاء لتوضيح البنية المنطقية للإجرائية، للتعابير، ولقوائم الوسطاء. لمزيد من التفاصيل، انظر الفصل 31 "التنسيق والأسلوب".
- تحقق من توثيق الإجرائية. تحقق من أن تحويل الشفرة الزائفة إلى تعليقات لا يزال دقيقاً. تحقق من توصيفات الخوارزمية، من التوثيق المتعلق بافتراضات الواجهة والتبعيات غير الواضحة، ومن تبرير ممارسات الترميز غير الواضحة. وإلى أخره. لمزيد من التفاصيل، انظر الفصل 32 "الشفرة ذاتية التوثيق".
- أزل التعليقات الإضافية. في بعض الأحيان، يتحول تعليق الشفرة الزائفة إلى تعليق زائد لا حاجة له مع النص البرمجي الذي يصفه، وخاصةً عندما يتم تطبيق عملية برمجة الشفرة الزائفة بشكل متكرر، حيث عندها التعليق يبدو فقط كجملة سابقة لإجرائية مُسماة بشكل جيد.

كرر حسب الحاجة

إذا كانت جودة الإجرائية ضعيفة، عُد إلى الشفرة الزائفة. إن البرمجة عالية الجودة هي عملية تكرارية، لذلك لا تتردد بتكرار نشاطات البناء مرة ثانية.

4.9 بدائل عملية برمجة الشفرة الزائفة

من أجل توفير المال، إن عملية برمجة الشفرة الزائفة هي أفضل طريقة لتوليد الصفوف والإجرائيات. هنا سنعرض بعض الطرق البديلة، المنصوح بها من قبل الخبراء. يمكنك استخدام هذه الطرق كبدائل أو كمكملات لعملية برمجة الشفرة الزائفة.

تطوير الاختبار – أولاً. إن الاختبار – أولاً هو أسلوب تطوير شائع، فيه تكون حالات الاختبار مكتوبة بشكل مُسبق لأي نص برمجي. إن هذه الطريقة موصوفة بمزيد من التفصيل في القسم 22-2 "الاختبار أولاً أو الاختبار أخيراً". ومثال عن كتاب جيد حول البرمجة بالاختبار أولاً "التطوير القائم على الاختبار" (كينت بيك 2003) (Kent Beck's Test-Driven Development: By Example (Beck 2003).

إعادة بناء التعليمات البرمجية. إن إعادة بناء التعليمات البرمجية هي طريقة تطوير، يمكنك فيها تحسين النص البرمجي من خلال سلسلة من التحويلات الواقية الدلالية. حيث يستخدم المبرمجين نماذج نص برمجي سيء لتحديد أقسام النص البرمجي التي تحتاج إلى تحسين. يصف الفصل 24 "إعادة التصنيع" هذه الطريقة بالتفصيل، ومثال عن كتاب جيد عن هذا الموضوع "إعادة بناء التعليمات البرمجية: تحسين تصميم النص البرمجي الحالي" (مارتن فاولر 1999) (Martin Fowler's Refactoring: Improving the Design of Existing Code (Fowler 1999).

تصميم حسب العقد. إن التصميم حسب العقد هو طريقة تطوير، فيها يتم الأخذ بعين الاعتبار كل إجرائية للحصول على الشروط المسبقة والشروط اللاحقة. استخدم التأكيدات للتوثيق والتحقق من الشروط المسبقة والشروط اللاحقة. إن المصدر الأفضل للتصميم حسب العقد هو الكتاب "بناء البرمجيات غرضية التوجه" (برتراند مايرز 1997) (Bertrand Meyers's Object-Oriented Software Construction ((Meyer 1997).

القرصنة؟ يحاول بعض المبرمجين القرصنة بطريقتهم الخاصة باتجاه النص البرمجي العامل، بدلاً من استخدام طريقة منهجية مثل عملية برمجة الشفرة الزائفة. إذا وجدت نفسك قد انحصرت في الزاوية عند كتابة إجرائية وعليك البدء من جديد، هذا مؤشر على إمكانية عمل طريقة برمجة الشفرة الزائفة بشكل أفضل. إذا وجدت نفسك ضعت أثناء كتابة الإجرائية، هذا مؤشر أخر على إمكانية عمل طريقة برمجة الشفرة الزائفة بشكل أكثر فائدة. هل سبق لك أن نسيت ببساطة كتابة جزء من صف أو جزء من إجرائية؟ هذا من الصعب حدوثه إذا كنت تستخدم طريقة برمجة الشفرة الزائفة. إذا وجدت نفسك لا تعرف من أين تبدأ، هذه إشارة واضحة أن عملية برمجة الشفرة الزائفة ستجعل حياتك البرمجية أسهل.

لائحة اختبار1: عملية برمجة الشفرة الزائفة 2

- هل تحققت من استيفاء المتطلبات الأساسية؟
- هل عرّفت المشكلة التي سيحلّها الصف؟
- هل التصميم عالي المستوى واضح بما فيه الكفاية لإعطاء الصف وكل من إجرائياته اسم جيد؟
- هل فكرت بكيفية اختبار الصف وكل من إجرائياته؟
- هل فكرت بالكفاءة بالدرجة الأولى من ناحية الواجهات المستقرة والتنفيذات القابلة للقراءة، أو بالدرجة الاولى من ناحية موافقة المصادر المتاحة للاستخدام والسرعة المتاحة للاستخدام؟
- هل تفحصت المكتبات القياسية وغيرها من المكتبات بحثاً عن الإجرائيات والمكونات القابلة للتطبيق؟
- هل تفحصت كتب المراجع بحثاً عن الخوارزميات المساعدة؟
- هل قمت بتصميم كل إجرائية باستخدام الشفرة الزائفة المفصلّة؟
- هل قمت بتفحص الشفرة الزائفة عقلياً؟ هل هي سهلة الفهم؟
- هل انتبهت إلى التحذيرات التي تعيدك للتصميم (استخدام البيانات العامة، العمليات التي تناسب أفضل صف أخر أو إجرائية أخرى، وما إلى ذلك)؟

[1] cc2e. com/0943

[2] **إشارة مرجعية** الغرض من هذه القائمة هو التحقق فيما إذا كنت قد اتبعت مجموعة جيدة من الخطوات لتوليد الإجرائية. من اجل قائمة التحقق التي تُركز على جودة الإجرائية نفسها، انظر "إجرائيات عالية الجودة"-قائمة اتحقق – الفصل 7،

- هل قمت بدقة بترجمة الشفرة الزائفة إلى الشفرة؟
- هل طبقت عملية برمجة الشفرة الزائفة بشكل متكرر، بتفكيك الإجرائيات إلى إجرائيات أصغر عندما هناك حاجة لذلك؟
- هل وثقت الافتراضات كما قمت بها؟
- هل أزلت التعليقات التي تبدو أنها زائدة عن الحاجة؟
- هل قمت باختيار الأفضل من تكرارات متعددة، بدلاً من مجرد التوقف بعد دورة التكرار الاولى؟
- هل تفهم جيداً النص البرمجي الخاص بك؟ هل هو سهل للفهم؟

نقاط مفتاحية

- تميل عمليات بناء الصفوف وبناء الإجرائيات إلى أن تكون عملية تكرارية. هذا يبدو واضح، عندما تميل عملية بناء الإجرائيات المتخصصة إلى الرجوع للوراء إلى عملية تصميم الصف.
- يتطلب كتابة شفرة زائفة جيدة استخدام اللغة الإنكليزية القابلة للفهم، وتجنب الميزات الخاصة بلغة برمجة واحدة، والكتابة على مستوى الغاية (أي وصف ما يفعله التصميم بدلاً من كيف سيفعل ذلك).
- إن عملية برمجة الشفرة الزائفة هي أداة مفيدة للتصميم المفصّل، وتجعل عملية كتابة الشفرة أسهل. تُترجم الشفرة الزائفة بشكل مباشر في تعليقات، تأكد من أن التعليقات دقيقة ومفيدة.
- لا تستقر على أول تصميم تُفكر فيه. ابحث في الطرق المختلفة في الشفرة الزائفة، ومن ثم اختر الطريقة الافضل قبل ان تبدأ بكتابة الشفرة.
- تفحص عملك عند كل خطوة، واطلب المساعدة من الاخرين أيضاً لتفحصه. بهذه الطريقة، ستتمكن من اكتشاف الاخطاء على أقل مستوى من الكلفة.

القسم الثالث: المتغيرات

في هذا القسم:

الفصل العاشر: قضايا عامّة في استخدام المتغيرات

الفصل الحادي عشر: قوّة أسماء المتغيرات

الفصل الثاني عشر: أنواع البيانات الأساسية

الفصل الثالث عشر: أنواع البيانات غير العادية

الفصل 10

قضايا عامّة في استخدام المتغيرات

المحتويات[1]

مواضيع ذات صلة

- تسمية المتغيرات: الفصل 11
- أنواع البيانات الأساسيّة: الفصل 12
- أنواع البيانات غير العادية: الفصل 13
- تنسيق التصريحات عن البيانات: "تخطيط التصريحات عن البيانات" في القسم 5.31
- توثيق المتغيرات: "توثيق التصريحات عن البيانات" في القسم 5.32

إنه لمن الطبيعي والمرغوب فيه في عملية البناء، ملء الفجوات الصغيرة في المتطلبات والهيكلة. وسيكون من غير الفعّال رسم مخططات لمستوى مجهري، مُفصّل بشكل كامل. يصف هذا الفصل قضية هيكلة البراغي والصواميل: تفاصيل التفاصيل لاستخدام المتغيرات. إذا لم تكن مُتأكداً من كفاءتك كـ "مبرمج خبير"، اقرأ "اختبار الإلمام في كتابة وقراءة البيانات" في القسم التالي، واعرف مستواك كمبرمج.

[1] cc2e.com/1085

يجب أن تكون المعلومات في هذا الفصل ذات قيمة خاصّة لك، إذا كنت مبرمج ذو خبرة. من السهل البدء باستخدام الممارسات العملية الخطيرة قبل معرفتك بالبدائل الممكنة، ومن ثم الاستمرار باستخدامها من غير العادة، حتى بعد تعلمك لطرق تجنبها. من الممكن أن يجد مبرمج خبير المناقشات حول زمن الارتباط في القسم 6.10 والمناقشات حول استخدام كل متغير لغرض معين في القسم 8.10 ممتعة بطريقة خاصّة.

خلال هذا الفصل، استخدم الكلمة "مُتغير" للإشارة إلى كائن، وكذلك للإشارة لأنواع البيانات المُضمنة مثل الأعداد الصحيحة " integers" والمصفوفات " arrays". تُشير عموماً العبارة "نوع البيانات" إلى أنواع البيانات المُضمّنة، بينما تُشير الكلمة "بيانات" إما إلى الكائنات أو إلى الأنواع المُضمّنة.

1.10 الإلمام بكتابة وقراءة البيانات



إن الخطوة الأولى في إنشاء بيانات فعّالة، هي في معرفة ما هو نوع البيانات لإنشائه. والمخزون الجيد من أنواع البيانات هو جزء أساسي من أدوات المُبرمج. وشرح أنواع البيانات هو خارج مجال هذا الكتاب، ولكن سيساعدك "اختبار الإلمام بكتابة وقراءة البيانات" في تحديد كم يلزمك لتتعلم هذه الأنواع.

اختبار الإلمام بكتابة وقراءة البيانات

ضع 1 بجانب كل مصطلح، يبدو مألوف بالنسبة إليك. إذا كنت تفكر أنك تعرف ماذا يعني مصطلح ما، ولكنك لست مُتأكد، ضع لنفسك 0.5. اجمع النقاط عندما تكون جاهزاً، وفسّر درجاتك وفقا لجدول النقاط في الأسفل.

المصطلح	النقاط	المصطلح	النقاط
بيان حرفي (literal)	-----------	نوع البيانات المجردة	-----------
متغير محلي	-----------	مصفوفة	-----------
جدول البحث	-----------	خريطة الأرقام الثنائية (bitmap)	-----------
بيانات العنصر (member data)	-----------	متغير بوليني ذو قيمتين فقط (boolean)	-----------
المؤشر	-----------	شجرة B (B-tree)	-----------
خاص (private)	-----------	متغير المحرف (character variable)	-----------
المشبك بأثر رجعي (retroactive synapse)	-----------	الصف الحاوي	-----------

التكامل المرجعي[1] (referential integrity)	-----------	الدقة المزدوجة	-----------
المُكدس (stack)	-----------	تدفق ممدود (elongated stream)	-----------
سلسلة محرفية (string)	-----------	نوع متعدد (enumerated type)	-----------
متغير منظم	-----------	النقطة العائمة	-----------
شجرة	-----------	الكومة (heap)	-----------
العامل typedef	-----------	الفهرس	-----------
الاتحاد (union)	-----------	العدد الصحيح	-----------
سلسلة القيم	-----------	القائمة المرتبطة	-----------
مختلف (variant)	-----------	المُسمى بثابت	-----------
مجموع النقاط	-----------		-----------

فيما يلي تستطيع أن تفسر النتيجة (بكامل الحرية):

0–14 أنت مبرمج مبتدئ، من المحتمل إنها سنتك الأولى من علوم الحاسوب في المدرسة أو أنك تُعلّم نفسك لغة البرمجة الأولى. يمكنك أن تتعلم المزيد من خلال قراءة واحدة من الكتب المذكورة في القسم الفرعي التالي. إن العديد من وصفات التقنيات المستخدمة في هذا الجزء من الكتاب، هي موجهة إلى المبرمجين المتقدمين، وستتعلم المزيد منها بعد قراءتك لواحدة من هذه الكتب.

15–19 أنت مبرمج متوسط، أو أنك مبرمج ذو خبرة، قد نسى العديد من الأشياء. على الرغم من أنه ستكون الكثير من هذه المفاهيم مألوفة بالنسبة إليك، فإنك تستطيع الاستفادة من قراءة واحدة من الكتب المذكورة تحت.

20–24 أنت مبرمج خبير. ومن الممكن أن يكون لديك هذه الكتب المذكورة تحت على الرف الخاص بك.

25–29 أنت تعرف أكثر مما أعرف أنا عن أنواع البيانات. أنصحك بكتابة كتابك الخاص عن علوم الحاسوب (ارسل إلي نسخة منه).

---

[1] **هامش**: retroactive legislation هو القانون الذي يدخل حيز التنفيذ قبل تاريخ صدوره (له أثر رجعي)، وبالتالي retroactive synapse الوصلة التي يعبرها أشياء سابقة لها بالوجود!

32–30 أنت محتال. إن المصطلحات " تدفق ممدود (elongated stream)"،" المشبك بأثر رجعي (retroactive synapse) "، و" سلسلة القيم" لا تُشير إلى أنواع البيانات- لقد قُمت باختلاقهم. من فضلك اقرأ القسم "الأمانة الفكرية" في الفصل 33،" الطابع الشخصي".

مصادر إضافية عن أنواع البيانات

هذه الكتب هي عبارة عن مصادر جيدة للمعلومات حول أنواع البيانات:

كورمن، ح. توماس، تشارلز أ. ليسرسون، رونالد ل. ريفيست. مُقدمة في الخوارزميات " Introduction to Algorithms". نيويورك، نيويورك: مغراو هيل 1990..

Ronald L. Rivest. Introduction to ،Charles E. Leiserson ،H. Thomas ،"Cormen Algorithms.
NY: McGraw Hill. 1990. ،New York

سيدجويك، روبرت. خوارزميات في سي++ "Algorithms in C++"، الأجزاء 1-4، الإصدار الثالث. بوسطن، ماساتشوستس: أديسون-ويسلي، 1998.

،MA: Addison-Wesley ،d ed. Boston3 ،Parts 1-4 ،Robert. Algorithms in C++ ،Sedgewick .1998

سيدجويك، روبرت. خوارزميات في سي++ "Algorithms in C++"، الجزء 5، الإصدار الثالث. بوسطن، ماساتشوستس: أديسون-ويسلي، 2002.

،MA: Addison-Wesley ،d ed. Boston3 ،Part 5 ،Robert. Algorithms in C++ ،Sedgewick .2002

2.10 جعل عملية التصريح عن المتغيرات سهلة

يصف هذا القسم كيفية تبسيط مهمة التصريح عن المتغيرات[1]. كُن متأكد، إنها مهمة صغيرة، ومن الممكن أن تفكر أنها أصغر من أن يُفرز جزء خاص بها في هذا الكتاب. ومع ذلك، أنت تصرف المزيد من الوقت في عملية إنشاء المتغيرات، ومن الممكن لتطوير العادات الصحيحة أن يوفر هذا عليك الوقت والإحباط على مدى حياة مشروع ما.

[1] **إشارة مرجعية:** لمزيد من التفاصيل حول تصميم تصاريح المتغير، انظر " تنسيق التصريحات عن البيانات " في القسم 5.31. لمزيد من التفاصيل حول عملية توثيقهم، انظر "توثيق التصريحات عن البيانات" في القسم 5.32

التصاريح الضمنية

تملك بعض لغات البرمجة التصاريح الضمنية للمتغير. على سبيل المثال، إذا كنت تستخدم متغير في لغة البرمجة فيجوال بيسيك، بدون التصريح عنه، فإن المترجم البرمجي سيقوم بالتصريح عنه بشكل أتوماتيكي بدلاً عنك (وذلك حسب إعدادات المُترجم عندك).

إن التصريح الضمني لمتغير، هو واحد من أخطر المميزات المتاحة في أية لغة برمجة. إذا كان برنامجك مكتوب بلغة البرمجة فيجوال بيسيك، فإنك تعرف مدى الإحباط في محاولة تفسير لماذا لا يملك المتغير `acctNo` القيمة الصحيحة، ومن ثم ستلاحظ ان `acctNum` هو المتغير الذي تمت إعادة تهيئته بالقيمة صفر. من السهل ارتكاب أخطاء كهذه، إذا كانت لغة البرمجة التي تستخدمها لا تتطلب التصريح عن المتغيرات.

إذا كنت تبرمج في لغة برمجة تتطلب منك أن تصرح عن المتغيرات، فسوف ترتكب خطأين قبل أن يقوم البرنامج بإزعاجك. في البداية عليك أن تضع كلا المتغيرين `acctNum` و`acctNo` داخل جسم الإجرائية. ومن ثم عليك أن تصرح عن كلا المتغيرين داخل الإجرائية. إن هذه مشكلة صعبة الارتكاب، وتُلغي عملياً مشكلة المتغيرات المترادفة. وإن اللغات التي تتطلب منك التصريح عن البيانات بشكل صريح، هي في جوهرها تتطلب استخدام البيانات بحذر أكبر، وهذه واحدة من ميزاتها الأساسية. ماذا تفعل إذا كنت تبرمج باستخدام لغة برمجة مع تصاريح ضمنية؟ فيما يلي بعض الاقتراحات:

أوقف التصاريح الضمنية. تسمح لك بعض المترجمات البرمجية بإلغاء تفعيل التصاريح الضمنية. على سبيل المثال، في لغة البرمجة فيجوال بيسيك، من الممكن أن تستخدم الخيار الصريح *`Option Explicit`*، الذي يجبرك على التصريح عن المتغيرات قبل استخدامها.

صرّح عن كل المتغيرات. بمجرد كتابتك لمتغير جديد، صرّح عنه، حتى وإن لم يتطلب المترجم البرمجي ذلك. هذا لن يتفادى كل الأخطاء، ولكنك سيمسك ببعضها.

استخدام اصطلاحات التسمية.[1] أنشئ اصطلاح تسمية للواحق المشتركة مثل `Num` و`No`، بهذه الطريقة لن تستخدم متغيرين، عندما يجب فقط استخدام واحد منهم.

تفحص أسماء المتغير. استخدم قائمة الإشارات المرجعية المولدة من قبل المترجم البرمجي أو من قبل أية وحدة برمجة أخرى. تضع العديد من المترجمات البرمجية المتغيرات الموجودة في الإجرائية في قائمة، سامحة لك بذلك بإلقاء الضوء على كلا المتغيرين `acctNum` و`acctNo`. وهذه القائمة تُلقي الضوء أيضاً على المتغيرات المُصرح عنها، ولكنها غير مُستخدمة.

[1] **إشارة مرجعية:** لمزيد من التفاصيل حول توحيد الاختصارات، انظر " المبادئ التوجيهية العامة للاختصارات" في القسم 11.6.

3.10 إرشادات لتهيئة المتغيرات



إن تهيئة البيانات غير الصحيحة، هي واحدة من أعظم مصادر توليد الخطأ في برمجة الحاسوب. تستطيع عملية تطوير التقنيات الفعالة لتجنب مشاكل التهيئة توفير الكثير من زمن التصحيح.

تنبع مشاكل التهيئة غير الصحيحة من احتواء المتغير على قيمة أولية، التي لا تتوقع احتواءه عليها. قد يحدث هذا بسبب العديد من الأسباب:

- لم يتم أبداً إســـناد قيمة للمتغير. قيمة المتغير هي أية بتات صـــادف وجودها في منطقة الذاكرة عند بدء البرنامج.[1]
- إن القيمة الموجودة في المتغير قديمة انتهت صـــلاحيتها. لقد تم اســـناد قيمة إلى المتغير في أحد النقاط، ولكن هذه القيمة لم تُعد صالحة.
- لقد تم إسناد قيمة لجزء من المتغير، أما الجزء الآخر فلم تتم إسناد قيمة إليه.

لدى السبب الأخير العديد من الاختلافات. حيث يمكنك تهيئة بعض من عناصر كائن، ولكن ليس كل العناصر. من الممكن أن تنسى تخصيص ذاكرة، ومن ثم تُهيئ "المتغير"، الذي يُشير إلى مؤشر غير مُهيئ. هذا يعني، أنك بالوقع تقوم باختيار جزء عشوائي من ذاكرة الحاسوب، وتُسند إليه قيمه ما. ومن الممكن أن تكون هذه عبارة عن ذاكرة تحوي بيانات. ومن الممكن أن تكون عبارة عن ذاكرة تحوي النص البرمجي. ومن يمكن أن يكون هذا النص البرمجي عبارة عن نظام التشغيل. فيمكن أن تظهر أعراض مشكلة المؤشر نفسها بطرق مدهشة تماما، تختلف في كل مرة.

فيما يلي إرشادات توجيهية لتجنب مشاكل التهيئة:

هيئ كل متغير، حالما يتم التصريح عنه. إن تهيئة المتغيرات بمجر

د التصريح عنها، هي عبارة عن صيغة غير مُكلفة للبرمجة الدفاعية. وهي عبارة عن بوصلة تأمين من أخطاء التهيئة. يُؤكّد المثال في الأسفل، بأنه سيتم إعادة تهيئة المتغير `studentGrades` في كل مرة تقوم باستدعاء الإجرائية التي تحوي هذا المتغير.

مثال بلغة البرمجة سي++ للتهيئة أثناء التصريح عن المتغير

```
float studentGrades[ MAX_STUDENTS ] = { 0.0 };
```

[1] **إشارة مرجعية:** للاطلاع على نهج الاختبار المُعتمد على تهيئة البيانات واستخدام الانماط، انظر " اختبار تدفق البيانات " في القسم 3.22.

قُم بتهيئة كل متغير في المكان القريب من مكان استخدامه لأول مرة. [1]لا تدعم بعض لغات البرمجة، بما فيها الفيجوال بيسيك، تهيئة المتغيرات عند التصريح عنها. من الممكن أن يؤدي هذا إلى أسلوب كتابة شفرة كما في التالي، والذي فيه تم تجميع التصاريح مع بعضها، ومن ثم تم تجميع عمليات التهيئة مع بعضها- كل ذلك بعيد عن الاستخدام الأول للمتغيرات.



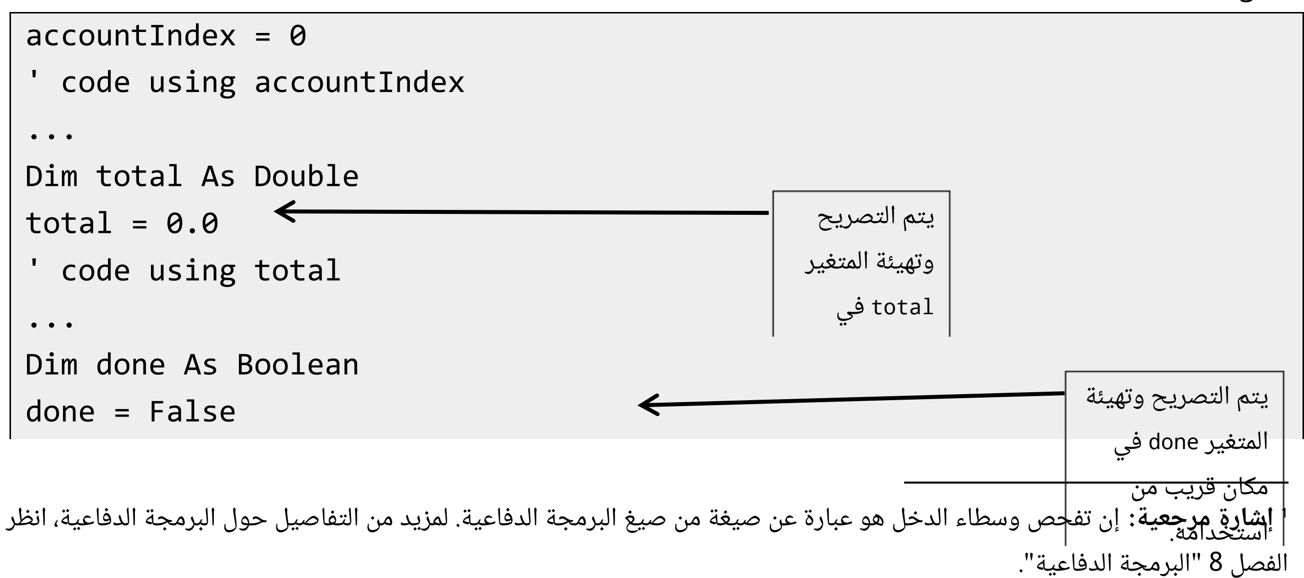
مثال بلغة البرمجة فيجوال بيسيك عن عملية تهيئة سيئة.

```
' declare all variables
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean
' initialize all variables
accountIndex = 0
total = 0.0
done = False
...
' code using accountIndex
...
' code using total
...
' code using done
While Not done
...
```

ممارسة عملية أفضل هي تهيئة المتغيرات أقرب ما يمكن لمكان استخدامها الأول:

مثال بلغة البرمجة فيجوال بيسك عن عملية تهيئة جيدة.

```
Dim accountIndex As Integer
```



```
accountIndex = 0
' code using accountIndex
...
Dim total As Double
total = 0.0
' code using total
...
Dim done As Boolean
done = False
```


[1] **إشارة مرجعية:** إن تفحص وسطاء الدخل هو عبارة عن صيغة من صيغ البرمجة الدفاعية. لمزيد من التفاصيل حول البرمجة الدفاعية، انظر الفصل 8 "البرمجة الدفاعية".

```
' code using done
While Not done
...
```

إن المثال الثاني متفوق على المثال الأول لعدة أسباب. في الوقت الذي يصل تنفيذ المثال الأول إلى الشفرة التي تستخدم"done". من الممكن أن تكون "done" قد عُدّلت. إذا لم تكن هذه هي الحالة عندما تكتب البرنامج لأول مرة، فمن الممكن أن تكون عند إجراء التعديلات اللاحقة. وأيضاً توجد مشكلة أخرى مع النهج الأول، وهي أن إلقاء جميع عمليات التهيئة مع بعضها يعطي الانطباع بأن كل المتغيرات استخدمت على طول الإجرائية، بينما في الحقيقة ""done مستخدمة فقط في النهاية.

أخيراً، نظراً لتعديل البرنامج (لو كان فقط من خلال التنقيح)، يمكن أن تُبنى الحلقات حول الشفرة التي تستخدمdone، وستكون done بحاجة إلى إعادة تهيئة. في المثال الثاني ستتطلب الشفرة تعديل صغير في مثل هذه الحالة. وبالتالي فإن الشفرة في المثال الأول أكثر عرضة لإنتاج خطأ تهيئة مُزعج.

هذا مثال عن مبدأ القُرب: المحافظة على النشاطات المرتبطة مع بعضها البعض.[1] يُطبق المبدأ نفسه على المحافظة على التعليقات قريبة من الشفرة التي تصفها، والمحافظة على شفرة إعداد الحلقة قريبه من الحلقة، وتجميع العبارات في شفرة خطيّة، وإلى العديد من المجالات الأخرى.

بشكل مثالي، صرّح وعرّف عن كل متغير في مكان قريب لمكان استخدامه الأول. يُنشئ التصريح نوع المتغير. يسند التعريف القيمة المحددة إلى المتغير. في لغات البرمجة التي تدعم هذا، مثل سي++ وجافا، يجب أن يتم التصريح عن المتغيرات وتعريفها في مكان قريب لمكان استخدامها أول مرة. بشكل مثالي، يجب أن يتم تعريف كل متغير في نفس وقت التصريح عنه، كما هو موضح فيما يلي:

مثال بلغة البرمجة جافا لعملية تهيئة جيدة.

```
int accountIndex = 0;
// code using accountIndex
...
double total = 0.0;
// code using total
...
boolean done = false;
// code using done
while ( ! done ) {
...
```

تمت تهيئة المتغير total في مكان قريب من مكان استخدامه

تمت تهيئة المتغير done في مكان قريب من مكان استخدامه.

[1] إشارة مرجعية: لمزيد من التفاصيل حول المحافظة على النشاطات المرتبطة مع بعض، انظر القسم 4.10 "النطاق".

استخدم final أو const إذا كان هذا ممكن. [1]من خلال التصريح عن متغير ليكون متغير نهائي final في لغة البرمجة جافا، أو ثابت const في لغة البرمجة سي++، تستطيع منع إسناد قيمه للمتغير بعد تهيئته. إن الكلمات المفتاحية const وfinal مفيدة في تعريف ثوابت الصف، ووسائط الدخل فقط، وأية متغيرات محلية مُراد لقيمها أن تحافظ على عدم التغيير بعد تهيئتها.

انتبه بشكل خاص إلى العدادات والمراكمات. إن المتغيرات i، j، k، sum، وtotal هي في أغلب الأوقات عدادات أو مراكمات. إن نسيان إعادة تعيين عداد أو مراكم قبل استخدامه في المرة القادمة، هو خطأ شائع. هيّئ بيانات عنصر الصف في الباني الخاص بالصف. تماماً كما يجب تهيئة متغيرات إجرائية داخل كل إجرائية، يجب أن تُهيّئ بيانات الصف داخل الباني الخاص به. إذا تم تخصيص الذاكرة في الباني، يجب أن يتم تحريرها في الهادم.

هيّئ بيانات أعضاء الصف في البواني الخاصة بها مثلما يجب تهيئة متغيرات الإجرائية داخل كل إجرائية، يجب تهيئة بيانات الصف داخل الباني الخاص بها. إذا خُصصت الذاكرة في الباني، فيجب أن تحرّر في الهادم.

تحقق فيما إذا كان هناك حاجة لإعادة التهيئة . اسأل نفسك فيما إذا كان المتغير سيحتاج إلى إعادة التهيئة، إما بسبب وجود حلقة في الإجرائية، تستخدم المتغير عدة مرات، أو بسبب إعادة المتغير لقيمته بين استدعاءات الإجرائية، ويحتاج إلى إعادة التعيين بين الاستدعاءات. إذا كانت هناك حاجة لإعادة التهيئة، تأكد من أن عبارة التهيئة داخل الجزء من الشفرة، الذي يتم تكراره.

هيئ الثوابت المُسماة مرة واحدة؛ هيأ المتغيرات مع التعليمات التنفيذية القابلة للتنفيذ. إذا كنت تستخدم متغيرات لمحاكاة الثوابت المُسماة، فلا بأس بكتابة الشفرة التي تُهيئهم مرة واحدة، في بداية البرنامج. للقيام بهذا، هيئهم في إجرائية ()Startup. هيئ المتغيرات الحقيقية في شفرة قابلة للتنفيذ، قريبه من مكان استخدامها. إن إحدى أكثر تعديلات البرنامج الشائعة، هي بتغيير الإجرائية، التي يتم استدعائها في الأصل مرة واحدة، لتتمكن من استدعائها أكثر من مرة. إن المتغيرات التي يتم تهيئتها في إجرائية على مستوى البرنامج ()Startup، لا يتم إعادة تهيئتها مرة ثانية داخل الإجرائية.

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول المحافظة على النشاطات المرتبطة مع بعضها، انظر القسم 14.2 "العبارات، التي ترتيبها لا يشكل مشكلة "

استخدم إعدادات المترجم البرمجي، التي تقوم بشكل أتوماتيكي بتهيئة كل المتغيرات. إذا كان المترجم البرمجي لديك يدعم خيار كهذا، فإن وجود إعداد للمترجم لتهيئة كل المتغيرات بشكل أتوماتيكي، هو خيار سهل حول موضوع الاعتماد على المترجم البرمجي. يمكن أن يُسبب الاعتماد على إعدادات خاصة لمترجم برمجي، على كل حال، مشاكل عندما تُنقل الشفرة إلى جهاز آخر ومترجم برمجي آخر. تأكد من توثيق استخدامك لإعدادات المترجم البرمجي؛ وإلا فمن الصعب الكشف على الافتراضات التي تعتمد على إعدادات محددة للمترجم البرمجي.

استفد من رسائل تحذير المترجم البرمجي الخاص بك. يُحذر العديد من المترجمات البرمجية عن استخدام متغيرات غير مُهيأة.

تفحص صلاحية وسطاء الدخل[1]. صيغة أخرى ذو قيمة للتهيئة هي بالتحقق من صلاحية وسطاء الدخل. قبل اسناد قيم الدخل إلى أي شيء، تحقق من معقولية هذه القيم.

استخدم متفحص وصول للذاكرة، وذلك لتفحص المؤشرات السيئة. في بعض أنظمة التشغيل، تتفحص شفرة برنامج التشغيل مرجعيات المؤشر الصالحة. في أنظمة تشغيل أخرى، يجب القيام بذلك بنفسك. على كل حال، ليس عليك أن تبقى على الخيار الخاص بك، لأنك تستطيع شراء متفحصات وصول للذاكرة، لتقوم بتفحص عمليات المؤشر لبرنامجك.

هيئ الذاكرة العاملة عند بداية برنامجك. يُساعد تهيئة الذاكرة العاملة إلى قيمة معروفة، على إظهار مشاكل التهيئة. يمكنك استخدام أيّة طريقة من الطرق التالية:

- تستطيع استخدام برنامج مالئ ذاكرة، لملء الذاكرة بقيم متوقعة. إن القيمة 0 مفيدة لبعض الأغراض، لأنها تؤكد على أن المؤشـرات غير المهيأة تؤشـر على ذاكرة منخفضـة، جاعلة هذه المؤشـرات سـهلة الكشـف عند اسـتخدامها. في معالجات شـركة Intel، إن القيمة xCC0 هي قيمة جيدة للاسـتخدام، لأنها تمثل شـفرة الألة لنقطة توقف breakpoints؛ إذا كنت تشـغل الشـفرة داخل مصـحح وتحاول تنفيذ البيانات بدلاً من الشفرة، فستكون راغباً با ستخدام نقاط التوقف breakpoints. فائدة أخرى لا ستخدام القيمة xCC0، هي بسـهولة التعرف أثناء عمليات تفريغ الذاكرة- نادرا ما تسـتخدم لأسـباب مشـروعة. كحل بديل، اقترح بريان كيرنيغان وروب بايك اسـتخدام الثابت xDEADBEEF0 كمالئ ذاكرة، وذلك لسهولة عملية التعرف في المُصحح (1999).

[1] **إشارة مرجعية:** للمزيد حول التحقق من وسطاء الدخل، انظر القسم 8.1 "حماية برنامجك من المدخلات غير المتاحة" وباقي الفصل 8 "البرمجة الدفاعية".

- إذا كنت تستخدم مالئ ذاكرة، فإنك تستطيع تغيير القيمة التي تستخدمها لملء الذاكرة مرة واحدة في لحظة. يكشف هذا البرنامج في بعض الأحيان عن مشاكل، تبقى مختبئة إذا لم تتغير أبداً البيئة الخلفية.
- من الممكن أن تملك برنامج يهيئ ذاكرته العاملة أثناء وقت الإقلاع. بينما الغرض من استخدام برنامج مالئ الذاكرة هو بإظهار العيوب، فإن الغرض من هذه التقنية هي بإخفاء العيوب. من خلال ملء الذاكرة العاملة بالقيمة نفسها في كل مرة، تستطيع أن تضمن بأن برنامجك لن يتأثر بالتغييرات العشوائية في ذاكرة الإقلاع.

10.4 النطاق

إن الـ "النطاق" هو طريقة للتفكير حول حالة النجومية للمتغير: كم هو مشهور؟ يُشير النطاق، الوضوح، على إلى مدى متغيراتك معروفة وكم يمكن الإشارة إليها في البرنامج. إن المتغير مع نطاق محدود أو صغير، معروف فقط في منطقة صغيرة من البرنامج- مثل فهرس حلقة مُستخدم فقط في حلقة صغيرة. أما متغير بنطاق كبير، معروف في عدة أماكن من البرنامج- على سبيل المثال، جدول معلومات الموظف، المُستخدم خلال البرنامج. تتعامل اللغات المختلفة مع النطاق بطرق مختلفة. في بعض اللغات البدائية، كل المتغيرات هي متغيرات عامة `global`. ولذلك لا تملك عندها أي تحكم على نطاق المتغير، وهذا يُنشأ العديد من المشاكل. في لغة البرمجة سي++، وفي اللغات المشابهة، يستطيع أن يكون المتغير مرئي إلى بلوك (قسم من الشفرة، مُضمّن داخل أقواس معقوفة)، أو لإجرائية، أو لصف (ومن الممكن لصفوفه المشتقة)، أو لكامل البرنامج. في لغة البرمجة جافا وسي#، يمكن للمتغير أن يكون مرئي لرزمة أو لمساحة الاسم (مجموعة من الصفوف).

توفر الأقسام التالية إرشادات توجيهية، مُطبقة على النطاق.

تمركز المراجع إلى المتغيرات

إن الشفرة بين المراجع لمتغير ما هي عبارة عن "نافذة (نقطة) ضعف". في النافذة من الممكن أن تضاف شفرة جديدة، أو تُبدّل قيمة المتغير من دون قصد، أو من الممكن أن ينسى قارئ الشفرة القيمة المفروض أن يحتويها المتغير. لذلك فإن تمركز المراجع إلى المتغيرات عن طريق المحافظة عليهم قريبين من بعضهم البعض هي فكرة جيدة دائما.

إن فكرة تمركز المراجع إلى متغير، هي بديهية جداً، ولكنها أيضاً فكرة تتيح إمكانية القياس الرسمي. إن إحدى طرق قياس مدى قرب مراجع لمتغير من بعضها، هي بحساب "امتداد" متغير. فيما يلي مثال:

مثال بلغة البرمجة جافا عن امتداد متغير

```
a = 0;
b = 0;
```

```
c = 0;
a = b + c;
```

في هذه الحالة، يوجد سطرين بين المرجع الأول للمتغير a وبين المرجع الثاني له، لذلك لديه امتداد يساوي 2. ويوجد فقط سطر واحد بين المرجعين للمتغير b، ولهذا يملك المتغير b امتداد يساوي 1، ويملك المتغير c امتداد يساوي 0. فيما يلي مثال ثاني:

مثال بلغة البرمجة جافا عن امتداد بين 1 و0

```
a = 0;
b = 0;
c = 0;
b = a + 1;
b = b / c;
```

في هذه الحالة،[1] يوجد سطر واحد بين المرجع الأول للمتغير b والمرجع الثاني له، والامتداد يساوي 1. ولا يوجد أية أسطر بين المرجع الثاني للمتغير b والمرجع الثالث له، والامتداد يساوي 0.

لقد تم حساب الامتداد المتوسط بحساب القيمة المتوسطة للامتدادات المنفصلة. في المثال الثاني، من أجل المتغير b، نتيجة (0+1)/2 هو القيمة المتوسطة للامتداد ويساوي 0.5. عندما تُحافظ على المراجع للمتغيرات قريبة من بعضها، فإنك تمكن الشخص القارئ لنصك البرمجي من التركيز على قسم واحد في الوقت الواحد. إذا كانت المراجع منفصلة عن بعضها البعض، فإنك تجبر القارئ على القفز داخل البرنامج من مكان إلى مكان. وهكذا فإن الميزة الأساسية من المحافظة على المراجع للمتغيرات مع بعضها البعض، هي بتحسين قابلية قراءة البرنامج.

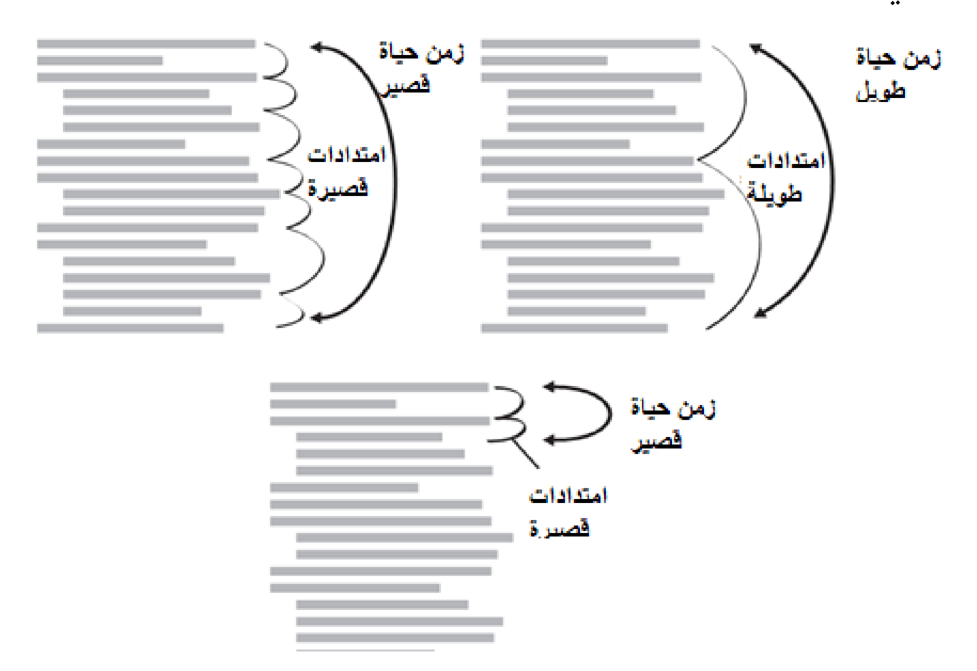
حافظ على "حياة" المتغيرات، لمدة قصيرة قدر الإمكان

إن المفهوم المتعلق بامتداد المتغير هو "زمن حياة" المتغير، وهو عدد العبارات البرمجية التي يعيشها متغير. تبدأ حياة متغير عند أول عبارة برمجية يتم فيها الإشارة إلى هذا المتغير، وتنتهي حياته عند أخر عبارة مرجعية، يتم فيها الإشارة إلى هذا المتغير.

بخلاف الامتداد، لا يتأثر زمن حياة المتغير بعدد مرات استخدام المتغير بين أول وأخر مرة، تمت الإشارة إليه. إذا تمت الإشارة إلى المتغير أول مرة على السطر الاول، وتمت الإشارة أخر مرة إليه على السطر 25، فإنه يملك زمن حياة يساوي 25 عبارة. إذا كانت هاتين السطرين هما السطرين الوحيدين، الذي تم فيهما استخدام المتغير، فإن هذا المتغير يملك امتداد متوسط يساوي 23 عبارة. أما لو تم استخدام المتغير على كل سطر من السطر 1

[1] **قراءة متعمقة**: لمزيد من المعلومات عن امتداد متغير، انظر مقاييس ونماذج هندسة البرمجيات (كونتي، دونزمور، وشين 1986).

إلى السطر 25، فإن المتغير كان سيملك امتداد متوسط يساوي 0 عبارة، ولكنه كان لا يزال يملك زمن حياة يساوي 25 عبارة. يوضح الشكل 1-10 كلا الامتداد وزمن الحياة.



الشكل 10-1 إن "زمن الحياة الطويل" يعني أن المتغير حي على مدى العديد من العبارات البرمجية. ويعني "زمن الحياة القصير" أن المتغير حي فقط على مدى عدة عبارات. يُشير "الامتداد" عن مدى قرب المراجع لمتغير من بعضها البعض.

كما في حالة الامتداد، الهدف من زمن الحياة هو المحافظة على قيمته صغيرة، والمحافظة على المتغير حي لمدة قصيرة قدر الإمكان. وكما في حالة الامتداد، إن الميزة الأساسية من المحافظة على قيمة زمن الحياة صغيره، هي بتقليل نافذة الضعف قدر الإمكان. فإنك بهذه الحالة تقلل من التبديل غير الصحيح أو غير المقصود لمتغير بين الأماكن التي تنوي تغييرها.

ميزة ثانية لإبقاء زمن حياة المتغير قصير، هي بأن هذا يُعطيك صورة دقيقة عن شفرتك. إذا كان هناك متغير، مُسند إليه قيمة في السطر 10 ولم يتم استخدامه حتى السطر 45، يشير هذا الفراغ الكبير بين المرجعين إلى أن المتغير مُستخدم بين السطور 10 و45. إذا تم اسناد قيمة للمتغير في السطر 44، وتم استخدامه في السطر 45، فإنه لا يوجد استخدامات ضمنية أخرى للمتغير، ويمكنك عندها التركيز على قسم صغير من الشفرة، عندما تقوم بالتفكير بذلك المتغير.

يقلل أيضاً زمن الحياة القصير من فرصة أخطاء التهيئة. عند قيامك بتعديل برنامج، فإنه تميل الشفرة الخطيّة إلى أن تتحول إلى حلقات، وأنت قد تميل إلى نسيان عمليات التهيئة التي تم القيام بها بعيداً عن الحلقة. أما بالمحافظة على شفرة التهيئة وشفرة الحلقة قريبين من بعضهم البعض، فإنك عندها تنقص فرصة قيام التعديلات بأخطاء تهيئة.

يجعل زمن الحياة القصير شفرتك أكثر قابلية للقراءة. كلما كان عدد السطور الذي يجب على القارئ التفكير فيها في الوقت الواحد، كلما كان من السهل فهم الشفرة. بطريقة مماثلة، فإنه كلما كان زمن الحياة أقصر، كلما قل

النص البرمجي الذي يجب المحافظة عليه على الشاشة، عندما تريد رؤية كل المراجع لمتغير خلال عمليات التعديل والتصحيح.

وأخيراً، إن أزمنة الحياة القصيرة مفيدة عند تفكيك إجرائية كبيرة إلى عدة إجراءات أصغر. إذا تمت المحافظة على المراجع لمتغير مع بعضها البعض، فإنه من السهل تفكيك الأقسام المتعلقة مع بعضها البعض من الشفرة إلى إجرائيات مستقلة.

قياس زمن الحياة لمتغير

يمكنك صياغة مفهوم زمن الحياة من خلال حساب عدد السطور بين أول وأخر مرجع إلى المتغير (بما فيها السطر الأول والسطر الأخير). فيما يلي مثال، فيه أزمنة الحياة طويلة جداً:

مثال بلغة البرمجة جافا عن متغيرات بأزمنة حياة طويلة بشكل مفرط.

```
1 // initialize all variables
2 recordIndex = 0;
3 total = 0;
4 done = false;
...
26 while ( recordIndex < recordCount ) {
27...
28 recordIndex = recordIndex + 1;
...
64 while ( !done ) {
...
69 if ( total > projectedTotal ) {
70 done = true;
```

أخر مرجع للمتغير recordIndex

أخر مرجع للمتغير total

أخر مرجع للمتغير done

فيما يلي أزمنة الحياة للمتغيرات في هذا المثال:

```
recordIndex   ( line 28 - line 2 + 1 ) = 27
total   ( line 69 - line 3 + 1 ) = 67
done   ( line 70 - line 4 + 1 ) = 67
Average Live Time   ( 27 + 67 + 67 ) / 3 ≈ 54
```

فيما يلي تمت إعادة كتابة المثال السابق، بطريقة تصبح فيها المراجع لمتغير مع بعضها البعض:

مثال بلغة البرمجة جافا عن متغيرات بأزمنة حياة جيدة وقصيرة

```
...
25 recordIndex = 0;
26 while ( recordIndex < recordCount ) {
```

تم تحريك تهيئة المتغير recordIndex للأسفل من السطر 3.

```
27...
28 recordIndex = recordIndex + 1;
...
62 total = 0;
63 done = false;
64 while ( !done ) {
...
69 if ( total > projectedTotal ) {
70 done = true;
```

تم تحريك تهيئة المتغير total والمتغير done للأسفل من السطور 4 و5.

فيما يلي أزمنة الحياة للمتغيرات في هذا المثال:

```
recordIndex ( line 28 - line 25 + 1 ) = 4
total ( line 69 - line 62 + 1 ) = 8
done ( line 70 - line 63 + 1 ) = 8
Average Live Time      ( 4 + 8 + 8 ) / 3 ≈ 7
```

بشكل بديهي، يبدو المثال الثاني أفضل من المثال الأول[1]، لأنه تم وضع عمليات التهيئة قريبة لمكان استخدام المتغيرات. إن الفرق المقاس لمتوسط زمن الحياة بين المثالين كبير: يوفر الفرق بين المتوسط لزمن الحياة في المثال الأول 54 والمتوسط في المثال الثاني 7، دعم كمّي جيد للأداء البديهي للمثال الثاني.

هل يفرّق العدد الثابت بين زمن حياة جيد وزمن حياة سيء؟ بين الامتداد الجيد والامتداد السيء؟ لم ينتج الباحثين حتى الأن هذه البيانات الكميّة، لكنه من الآمن افتراض أن تصغير الامتداد وزمن الحياة هو فكرة جيدة.

إذا كنت ترغب في تطبيق أفكار الامتداد وزمن الحياة على المتغيرات العامة، فستجد أن المتغيرات العامة تملك امتدادات وأزمنة حياة ضخمة – واحدة من الأسباب الجيدة العديدة لتجنب المتغيرات العامة.

إرشادات توجيهية عامة لتصغير النطاق

فيما يلي بعض الإرشادات التوجيهية الخاصة، التي يمكنك استخدامها لتصغير النطاق:

هيئ المتغيرات المستخدمة في الحلقة مباشرةً قبل الحلقة،2 بدلاً من العودة إلى بداية الإجرائية الحاوية على الحلقة. يُحسن القيام بهذا من فرصة تذكرك القيام بالتعديلات المقابلة لتهيئة الحلقة عند قيامك بتعديل الحلقة. لاحقاً، عندما تقوم بتعديل البرنامج وتضع حلقة أخرى حول الحلقة الاولية، عندها ستعمل التهيئة عند كل مرور عبر الحلقة الجديدة، بدلاً من فقط المرور الأول.

[1] **قراءة متعمقة**: لمزيد من المعلومات حول "حياة" المتغيرات، انظر مقاييس ونماذج هندسة البرمجيات (كونتي، دونزمور، وشين 1986).

[2] **إشارة مرجعية:** لمزيد من التفاصيل حول تهيئة المتغيرات قريباً من مكان استخدامها، انظر القسم 10.3،"إرشادات توجيهية لتهيئة المتغيرات"، سابقاً في هذا الفصل.

لا تسند قيمة إلى متغير حتى قبل استخدام القيمة مباشرةً.[1] من الممكن أن تكون اختبرت احباط محاولة التفكير فيما إذا تم استناد قيمة إلى متغير أو لا. وكل ما يمكنك القيام به هو توضيح المكان الذي يتلقى فيه المتغير قيمته، تدعم اللغات الجيدة مثل سي++ وجافا عمليات تهيئة للمتغير كما يلي:

مثال بلغة البرمجة سي++ لعمليات جيدة لتهيئة والتصريح عن متغير

```
int receiptIndex = 0;
float dailyReceipts = TodaysReceipts();
double totalReceipts = TotalReceipts( dailyReceipts );
```

جمّع العبارات البرمجية المرتبطة مع بعضها البعض.[2] تُظهر الامثلة التالية إجرائية لتلخيص الإيصالات اليومية، وتوضح كيفية وضع المراجع للمتغيرات مع بعضها البعض بحيث يسهل تحديد موقعها. يوضح المثال الأول انتهاك هذا المبدأ:

مثال بلغة البرمجة سي++ لاستخدام مجموعتين من المتغيرات بطريقة مربكة.

```
void SummarizeData(...) {
..
 GetOldData( oldData ،&numOldData );
 GetNewData( newData ،&numNewData );
 totalOldData = Sum( oldData ،numOldData );
 totalNewData = Sum( newData ،numNewData );
 PrintOldDataSummary( oldData ،totalOldData ،numOldData );
 PrintNewDataSummary( newData ،totalNewData ،numNewData );
 SaveOldDataSummary( totalOldData ،numOldData );
 SaveNewDataSummary( totalNewData ،numNewData );
...
}
```

عبارات برمجية تستخدم مجموعتين من المتغيرات

لاحظ هذا، في هذا المثال، عليك أن تحافظ على مسار المتغيرات التالية في وقت واحد *oldData*، و*newData*، و*numOldData*، و*numNewData*، و*totalOldData*، و*totalNewData* - ستة متغيرات من أجل فقط هذا المقطع القصير. يُظهر المثال التالي كيفية التخفيض من هذا العدد الكبير من المتغيرات إلى فقط ثلاث عناصر داخل كل بلوك من النص البرمجي:

[1] **إشارة مرجعية:** للمزيد حول هذا الأسلوب من التصريح والتصريح عن متغير، انظر "بشكل مثالي، صرح وعرّف عن كل متغير في مكان قريب لمكان استخدامه الأول" في القسم 10.3.

[2] **إشارة مرجعية:** لمزيد من التفاصيل حول المحافظة على العبارات المرتبطة مع بعضها البعض، انظر القسم 2.14 "العبارات التي لا يشكل ترتيبها مشكلة".

مثال بلغة البرمجة سي++ عن استخدام مجموعتين من المتغيرات بطريقة أكثر قابلة للفهم.

```
void SummarizeData(... ) {
 GetOldData( oldData ،&numOldData );
 totalOldData = Sum( oldData ،numOldData );
 PrintOldDataSummary( oldData ،totalOldData ،numOldData );
 SaveOldDataSummary( totalOldData ،numOldData );
...
 GetNewData( newData ،&numNewData );
 totalNewData = Sum( newData ،numNewData );
 PrintNewDataSummary( newData ،totalNewData ،numNewData );
 SaveNewDataSummary( totalNewData ،numNewData );
...
}
```

مجموعة العبارات البرمجية التي تستخدم .oldData

عندما يتم تقسيم الشفرة، فإن الكتلتين الناتجين أقصر من الكتلة" **block** " الأصلية، وبشكل مستقل يحتويان على عدد متغيرات أقل. هذين الكتلتين أسهل للفهم، وإذا احتجت إلى تقسيم هذه الشفرة إلى عدة إجرائيات منفصلة، فإن الكتل القصيرة ذات العدد القليل من المتغيرات ستتمكن من تطوير إجرائيات جيدة التعريف.

اكسر مجموعات العبارات البرمجية المرتبطة مع بعضها إلى عدة إجرائيات منفصلة. كل الأمور الأخرى متساوية، حيث يميل متغير داخل إجرائية قصيرة إلى امتلاك امتداد وزمن حياة أصغر من متغير في إجرائية أطول. من خلال كسر العبارات البرمجية المرتبطة إلى إجرائيات أصغر ومنفصلة، فإنك تُخفض من النطاق الذي يملكه المتغير.

ابدأ من مجالات الرؤية الاكثر تقييداً، ومن ثم وسّع نطاق المتغير فقط إذا كانت هناك حاجة.[1] إن المحافظة على المتغيرات محلية قدر الإمكان هو جزء من عملية تصغير نطاق المتغير. إنه من الأكثر صعوبة تخفيض نطاق متغير يملك نطاق كبير، من توسيع نطاق متغير يملك نطاق صغير- بكلمات أخرى، إن تحويل متغير عام إلى متغير صف أصعب من تحويل متغير صف إلى متغير عام. إن تحويل عنصر البيانات المحميّة إلى عنصر بيانات خاصة أصعب من عملية التحويل العكسية. لذلك السبب، عندما يكون هنا شك، فإنه مفضل النطاق الأصغر الممكن لمتغير: محلي لحلقة محددة، محلي لإجرائية مستقلة، ثم خاص لصف، ثم محمي، ثم رزمة (إذا كانت لغة البرمجة التي تستخدمها تدعم هذا)، وعام فقط كحل أخير.

[1] **إشارة مرجعية**: للمزيد حول المتغيرات العامة، انظر القسم 3.13 "البيانات العامة".

تعليقات حول تصغير النطاق

تعتمد العديد من الطرق البرمجية لتصغير نطاق المتغير على وجهات نظرها في قضايا "السهولة" و"الإدارة الفكرية". يجعل بعض المبرمجين كل المتغيرات عامة، لأن النطاق العام يجعل المتغيرات سهلة للوصول، وبالتالي لن يتوجب على المبرمجين التفكير بقائمة الوسطاء وقواعد النطاق للصف. برأيهم، تفوق سهولة القدرة على الوصول إلى المتغيرات من المخاطر التي تنطوي عليها.

يُشير المبرمجين الأخرين إلى متغيراتهم كنوع متغيرات محلية بقدر الإمكانية المتاحة لذلك لأن النطاق المحلي يساعد على الإدارة الفكرية[1]. كلما تمكنت من إخفاء المزيد من المعلومات، فهذا يعني أن كمية أقل عليك أن تحتفظ فيها في عقلك (تفكر فيها) في الوقت الواحد. وكلما قلّت الكمية التي يجب أن تحتفظ فيها في عقلك، كلما كانت الفرصة قليلة لارتكابك لخطأ بسبب نسيانك واحدة من عدة تفاصيل عليك أن تتذكرها.



يتلخص الفرق بين فلسفة "السهولة" وفلسفة "الإدارة الفكرية" إلى الفرق في التركيز بين كتابة البرامج وقراءتها. من الممكن أن يجعل تصغير النطاق، في الواقع، البرامج سهلة الكتابة، ولكن البرنامج الذي يكون فيه أية إجرائية تستخدم أية متغير في أي وقت، هو أصعب للفهم من برنامج يستخدم إجرائيات مُحلّلة بشكل جيد. في هكذا برنامج، لا يمكنك أن تفهم فقط إجرائية واحدة؛ بل عليك أن تفهم كل الإجرائيات التي تتشارك معها الإجرائية البيانات العامة. برامج كهذه صعبة القراءة، وصعبة التصحيح، وصعبة التعديل.

بناءً على ذلك، عليك أن تُصرّح عن كل متغير، ليكون مرئي إلى أصغر مقطع[2]، تحتاج أن تراه من الشفرة. وإذا تمكنت من اقتصار نطاق المتغير على حلقة مفردة أو إجرائية مفردة، فسيكون هذا عظيم. وإذا لم تتمكن من اقتصار النطاق على إجرائية واحدة، فقم بتقييد نطاق رؤية الإجرائية إلى صف واحد. وإذا لم تتمكن من تقييد نطاق المتغير إلى الصف الأكثر ارتباط مع المتغير، عندها قُم بإنشاء إجرائيات لمشاركة بيانات المتغير مع الصفوف الأخرى. سوف تجد أنه من النادر، إذا لم يكن من المستحيل أن تحتاج إلى استخدام البيانات العامة المجردة.

[1] **إشارة مرجعية:** إن فكرة تصغير النطاق متعلقة مع فكرة إخفاء المعلومات. لمزيد من التفاصيل، انظر "أسرار الإخفاء (إخفاء المعلومات)" في القسم 3.5.

[2] **إشارة مرجعية:** لمزيد من التفاصيل حول استخدام إجرائيات الوصول، انظر "استخدام إجرائيات الوصول بدلاً من البيانات العامة" في القسم 3.13.

5.10 الاستمرارية (زمن الاستمرار):

"الاستمرارية" هي كلمة أخرى لامتداد الحياة لقطعة من البيانات. تأخذ الاستمرارية العديد من الصيغ. تستمر بعض المتغيرات

- من أجل حياة كتلة مُحدّدة من شـفرة أو من إجرائية. إن المتغيرات المصـرح عنها داخل حلقة for في لغة البرمجة سي++ أو جافا هي أمثلة عن هذا النوع من الاستمرارية.
- طالما كنت تسمح لهم. في لغة البرمجة جافا، المتغيرات، التي تم انشائها هي مع ديمومة جديدة حتى يتم "جمع القمامة". أما في لغة البرمجة ســي++، فإن المتغيرات، التي تم انشـــائها هي مع ديمومة جديدة حتى يتم حذف المتغيرات.
- من أجل حياة برنامج. تُناســب المتغيرات العامة في معظم لغات البرمجة هذا الوصــف، كما تفعل المتغيرات الساكنة في لغة البرمجة سي++ وجافا.
- للأبد. من الممكن أن تتضــمن هذه المتغيرات قيم، تُخزنها في قاعدة بيانات بين عمليات تنفيذ برنامج. على سبيل المثال، إذا كان لديك برنامج تفاعلي، يستطيع فيه المستخدمين تغيير لون الشاشة، عندها تســتطيع أن تُخزن ألوان المســتخدمين في ملف، ومن ثم تعود وتقرئهم في كل مرة يتم فيها تحميل البرنامج.

تظهر المشكلة الأساسية مع الاستمرارية، عندما تفترض أن المتغير يملك استمرارية أكبر من تلك التي يملكها بالحقيقة. المتغير مثل إبريق حليب في الثلاجة. من المفترض أن يستمر أسبوع. في بعض الأحيان يستمر هذا شهر، وفي بعض الأحيان يتحمّض بعد خمسة أيام. لا يمكن التنبأ بالمتغير. إذا قمت باستخدام قيمة متغير بعد انتهاء امتداد حياته الطبيعية، فهل سيحتفظ بقيمته؟ في بعض الأحيان القيمة في المتغير حامضة، وأنت تعرف أن لديك خطأ. مما يتيح لك أن تتخيل أنك قد استخدمته بشكل صحيح.

فيما يلي بعض الخطوات، التي تساعدك على تجنب هذا النوع من المشاكل:

- اسـتخدم شـفرة التصـحيح أو المصـادقات في برنامجك للتحقق من منطقيّة قيم المتغيرات الحرجة[1]. إذا كانت القيم غير منطقية، أظهر رســالة تحذير تُخبرك عن ضــرورة البحث عن مكان التهيئة غير الصحيحة.
- ضع للمتغيرات "قيم غير منطقية" عندما تنهي منها. على سبيل المثال، من الممكن أن تسند إلى مؤشر القيمة null بعد حذفه.

[1] **إشارة مرجعية:** إن شفرة التصحيح هي سهلة التضمين في إجرائيات الوصول وتمت مناقشتها بشكل أكبر في "مزايا إجرائيات الوصول" في القسم 3.13.

- اكتب الشفرة التي تفترض أن البيانات غير مستمرة. على سبيل المثال، إذا كان لدى متغير قيمة معينة عندما تخرج من إجرائية، فلا تفترض أن لدى هذا المتغير القيمة نفســها في المرة القادمة التي تدخل فيها إلى الإجرائية نفســها. هذا لا يتم تطبيقه إذا كنت تســتخدم الميزات الخاصــة بلغة برمجة، التي تضمن بقاء قيمة المتغير نفسها، كما الميزة الساكنة *static* في لغة البرمجة سي++ وجافا.
- طوّر عادة التصريح وتهيئة البيانات تماماً قبل استخدامها. إذا وجدت بيانات مُستخدمة بدون عملية تهيئة قريبة، فيجب أن يرتابك الشك.

6.10 زمن الارتباط

موضوع تهيئة ذو آثار بعيدة المدى على صيانة البرامج وقابلية التعديل هو "زمن الارتباط": وهو الوقت الذي يكون فيه المتغير وقيمته مرتبطين مع بعضهما البعض (ثيمبلي 1988). هل هم مرتبطين مع بعضهما عندما يتم كتابة الشفرة؟ أو عندما يتم ترجمة الشفرة برمجياً (تجميعها)؟ أو عندما يتم تحميل الشفرة؟ أو عندما يتم تشغيل البرنامج؟ أو في أوقات أخرى؟

من الممكن أن يكون لصالحك استخدام أحدث وقت ممكن لارتباط. بشكل عام، كلما جعلت وقت الارتباط أحدث (في وقت لاحق)، كلما زادت المرونة التي تبنيها في نصك البرمجي. يُظهر المثال التالي الارتباط في أقرب وقت ممكن، عندما يتم كتابة الشفرة:

مثال بلغة البرمجة جافا لمتغير مُرتبط في وقت كتابة الشفرة

```
titleBar.color = 0xFF; // 0xFF is hex value for color blue
```

يتم ربط القيمة 0xFF للمتغير titleBar.color في وقت كتابة الشفرة، لأن القيمة 0xFF هي قيمة حرفية مرمزة بالترميز الصعب داخل البرنامج. إن الترميز الصعب مثل هذا، هو تقريباً دائماً فكرة سيئة، لأن القيمة 0xFF تتغير، حيث يمكنها الخروج من التزامن بالقيمة 0xFF المستخدمة في مكان آخر من التعليمات البرمجية التي يجب أن تكون لها نفس القيمة مثل هذه.

فيما يلي مثال عن الربط في وقت لاحق قليلا، عندما يتم ترجمة الشفرة برمجياً:

مثال بلغة البرمجة جافا لمتغير مرتبط في وقت الترجمة البرمجية (التجميع)

```
private static final int COLOR_BLUE = 0xFF;
private static final int TITLE_BAR_COLOR = COLOR_BLUE;
...
titleBar.color = TITLE_BAR_COLOR;
```

إن TITLE_BAR_COLOR هو ثابت مُسمّى، وهو تعبير يستبدل به المجمع البرمجي قيمتة عند زمن الترجمة البرمجية. وهو تقريباً دائماً أفضل من الترميز الصعب، إذا كانت لغة البرمجة المُستخدمة تدعم ذلك. فهذا يزيد

من قابلية القراءة لأن TITLE_BAR_COLOR يُخبرك بالمزيد عما يمثله أكثر مما يفعله 0xFF. يجعل هذا تغيير لون شريط العنوان أسهل، لأنه تغيير واحد ينسحب على كل الحالات. وهذا لا يتكبد عقوبة الأداء وقت التشغيل.

فيما يلي، مثال عن ارتباط لاحق، في وقت التشغيل:

مثال بلغة البرمجة جافا لمتغير مرتبط في وقت التشغيل

```
titleBar.color = ReadTitleBarColor();
```

إن ()ReadTitleBarColor إجرائية، تقرأ قيمة بينما يتم تنفيذ برنامج، من الممكن من ملف سجل في Microsoft Windows أو من ملف الخصائص لجافا.

هذه الشفرة أكثر قابلية للقراءة وأكثر مرونة مما ستكون عليه فيما لو تم ترميز القيمة باستخدام الترميز الصعب. تحتاج إلى تغيير البرنامج لتغيير المتغير titleBar.color (لون شريط العنوان)؛ ببساطة يمكنك تغيير محتوى المصدر الذي يُقرأ من قبل الإجرائية ()ReadTitleBarColor .يتم استخدام هذه الطريقة عادةً للتطبيقات التفاعلية، التي فيها يختار المستخدم بيئة التطبيق المناسبة.

لا يزال يوجد اختلاف آخر في وقت الار تباط، الذي يجب أن يتم العمل معه عندما يتم استدعاء الإجرائية ()ReadTitleBarColor .حيث كان بالإمكان استدعاء تلك الإجرائية في وقت تحميل البرنامج، أو في كل مرة يتم فيها إنشاء النافذة، أو في كل مرة يتم رسم فيها النافذة – ويمثل كل بديل منها على التسلسل أوقات ارتباط لاحقة.

للتلخيص، فيما يلي الأوقات، التي يمكن فيها لمتغير أن يُربط لقيمة في هذا المثال. وقد تختلف التفاصيل إلى حد ما في حالات أخرى.

- زمن كتابة الشفرة (استخدام الأرقام السحرية)
- زمن الترجمة البرمجية (استخدم الثابت المُسمى)
- زمن التحميل (قراءة قيمة من مصدر خارجي مثل ملف سجل النوافذ او ملف الخصائص لجافا)
- زمن تهيئة الكائن (مثل قراءة القيمة في كل مرة يتم فيها إنشاء نافذة)
- في الوقت المناسب (مثل قراءة القيمة في كل مرة يتم فيها رسم نافذة)

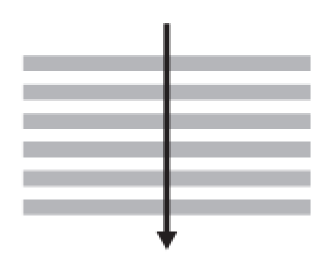
بشكل عام، كلما كان وقت الارتباط أبكر، كلما قلت المرونة والتعقيد. من أجل الخيارين الأولين، إن استخدام الثوابت المسماة مُفضل على استخدام الأرقام السحرية، وذلك للعديد من الأسباب، وهكذا يمكنك أن تحصل على المرونة التي توفرها الثوابت المسماة عن طريق استخدام الممارسات البرمجية الجيدة. أبعد من ذلك، كلما زادت المرونة المرغوب بها، كلما زاد تعقيد الشفرة المطلوب لدعم هكذا مرونة، وكلما أصبحت الشفرة أكثر عرضة للخطأ. لأنه تعتمد البرمجة الناجحة على تصغير التعقيد، لذلك سيبني المبرمج الخبير بقدر المرونة المطلوبة لمقابلة المتطلبات البرمجية، ولكن لن يقوم بإضافة مرونة- والتعقيد المرتبط بها- بما يتجاوز ما هو مطلوب.

7.10 العلاقة بين أنواع البيانات وهياكل التحكم

إن أنواع البيانات وبنى التحكم مرتبطة مع بعضها البعض بطرق مُعرّفة بشكل جيد، تم وصفها بالأصل من قبل عالم الحاسوب البريطاني مايكل جاكسون (جاكسون1975). هذا القسم يرسم العلاقة العادية بين البيانات والتحكم بالتدفق.

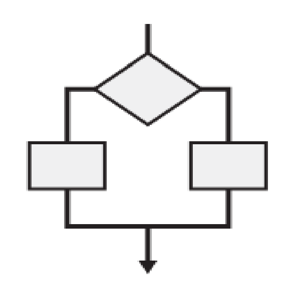
رسم جاكسون الاتصالات بين ثلاث أنواع من البيانات وبنى التحكم المقابلة لها:

تُترجم البيانات المتسلسلة إلى عبارات متسلسلة في برنامج[1]. تتألف السلاسل من عناقيد من البيانات المستخدمة مع بعضها بترتيب معين، كما هو مُقترح في الشكل 2.10. إذا كان لديك خمس عبارات برمجية في سطر، تتعامل مع خمس قيم مختلفة، فإنهم عبارة عن عبارات برمجية متسلسلة. إذا كنت تقرأ اسم عامل، أو رقم التأمين الاجتماعي، وعنوان، ورقم هاتف، وعمر من ملف، عندها فإنه لديك عبارات برمجية متسلسلة في برنامجك، الذي يقرأ البيانات المتسلسلة من ملف.



الشكل 10-2 البيانات المتسلسلة هي بيانات يتم معالجتها في ترتيب معين.

تُترجم البيانات الانتقائية إلى عبارات `if` وعبارات الحالة (switch) في برنامج[2]. بشكل عام، إن البيانات الانتقائية هي مجموعة، فيها واحدة من عدة قطع للبيانات مُستخدمة في زمن محدد، ولكن فقط قطعة واحدة، كما هو موضح في الشكل 10-3. يجب على العبارات البرمجية من البرنامج الموافقة أن تقوم بالاختيار الفعلي، وهذه العبارات تتألف من عبارات `if-then-else`، أو عبارات الحالة. إذا كان لديك برنامج الرواتب للموظفين، فمن الممكن أن تعالج الموظفين بشكل مختلف تماماً، وذلك بالاعتماد فيما إذا كان الدفع لهم حسب الساعة أو حسب الراتب. مرة ثانية، تتطابق الأنماط في الشفرة مع الانماط في البيانات.

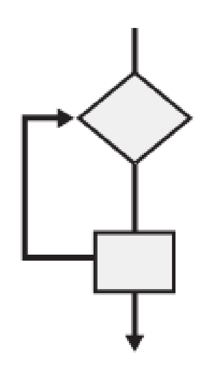


[1] **إشارة مرجعية:** للمزيد من التفاصيل حول السلاسل، انظر الفصل 14 " تنظيم شفرة خطيّة ".

[2] إشارة مرجعية: لمزيد من التفاصيل حول الجمل الشرطية، انظر الفصل 15 " استخدام الشرطيات".

الشكل 10-3 تسمح البيانات الانتقائية باستخدام قطعة واحدة أو القطعة الأخرى، ولكن ليس كلا القطعتين.

تُترجم البيانات التكرارية في هياكل الحلقات for، repeat، وwhile في برنامج.[1] يتم تكرار البيانات التكرارية من نفس نوع البيانات عدة مرات، كما هو مقترح في الشكل 10-4. بشكل نموذجي، يتم تخزين البيانات التكرارية كعنصر في عداد، أو تسجيل في ملف، أو عناصر في مصفوفة. من الممكن أن يكون لديك قائمة من أرقام التأمين الاجتماعي، التي تقرئها من ملف. ستتطابق البيانات التكرارية مع حلقة الشفرة التكرارية المستخدمة لقراءة البيانات.



الشكل 10-4 يتم تكرار البيانات التكرارية.

يمكن أن تكون بياناتك الحقيقية هي مجموعة من أنواع البيانات المتسلسلة والانتقائية والتكرارية. يمكن أن تُجمع كتل الارتباط البسيطة لوصف أنواع البيانات الأكثر تعقيداً.

8.10 استخدام كل متغير لغرض واحد بالضبط



من الممكن استخدام المتغيرات من أجل أكثر من غرض واحد في عدة طرق دقيقة. ولكن سيكون من الأفضل عدم استخدام هذا النوع من الطرق الدقيقة.

استخدم كل متغير لغرض واحد فقط. في بعض الأحيان من المغري استخدام متغير واحد في مكانين مختلفين من أجل نشاطين مختلفين. عادةً، يتم تسمية المتغير بشكل غير مناسب لواحدة من استخداماته كمتغير "مؤقت"، مُستخدم في كلا الحالتين (بالاسم العادي غير المفيد x أو temp). فيما لي مثال يُظهر متغير مؤقت، مُستخدم لغرضين:



مثال بلغة البرمجة سي++ لاستخدام متغير واحد لغرضين- ممارسة سيئة

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
temp = Sqrt( b*b - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
```

[1] **إشارة مرجعية:** لمزيد من التفاصيل حول الحلقات، انظر الفصل 16 "التحكم بالحلقات".

```
...
// swap the roots
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

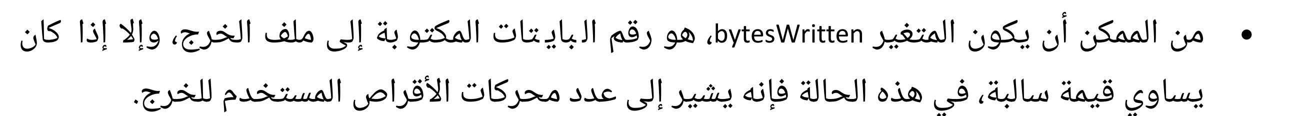
سؤال: ما هي العلاقة بين *temp* في بضعة السطور الاولى، وبين *temp* في بضعة السطور الأخيرة؟[1] الجواب: كلا الـ temp لا يملكان أية علاقة. قد يوحي استخدام المتغير نفسه في كلا الحالتين أنهما مرتبطين مع بعضهما، ولكن هذا غير صحيح. يجعل إنشاء متغيرات فريدة لكل غرض، الشفرة أكثر قابلية للقراءة. فيما يلي تحسين للمثال السابق:

مثال بلغة البرمجة سي++ لاستخدام متغيرين لغرضين مختلفين- ممارسة جيدة

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
discriminant = Sqrt( b*b - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
...
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

تجنّب المتغيرات ذات المعاني الخفية. طريقة أخرى لاستخدام متغير من أجل أكثر من غرض واحد، هي بامتلاك قيم مختلفة للمتغير، تعني أشياء مختلفة. على سبيل المثال:

- من الممكن أن تمثل القيمة في المتغير pageCount إلى عدد الصفحات المطبوعة، وإلا إذا كانت تساوي -1، في هذه الحالة فإنها تشير إلى حدوث خطأ.



- من الممكن أن يمثل المتغير customerId رقم العميل، وإلا إذا كان يساوي 500,000، فإنه عليك أن تقوم بعملية طرح 500,000 للحصول على رقم حساب الزبزن المقصر بالدفع.
- من الممكن أن يكون المتغير bytesWritten، هو رقم البايتات المكتوبة إلى ملف الخرج، وإلا إذا كان يساوي قيمة سالبة، في هذه الحالة فإنه يشير إلى عدد محركات الأقراص المستخدم للخرج.

[1] **إشارة مرجعية**: يجب أيضاً على وسطاء الإجرائية أن تكون مستخدمة لغرض واحد فقط، لمزيد من التفاصيل عن استخدام وسطاء الإجرائية، انظر القسم 5.7 "كيفية استخدام وسطاء الإجرائية".

تجنب المتغيرات مع هذه الأنواع من المعاني الخفية. الاسم التقني لهذا النوع من الخطأ هو "الاقتران المختلط" (بيج-جونز 1988). تم مد المتغير ليقوم بعملين، يعني هذا أن المتغير هو النوع الخاطئ لواحد من هذين العملين. في مثال الـ pageCount، يُشير المتغير pageCount بشكل طبيعي إلى عدد الصفحات؛ انه عدد صحيح. ولكن عندما pageCount يساوي 1- فإنه يُشير إلى حدوث خطأ؛ هنا يبدو العدد الصحيح كقيمة منطقية (صح أو خطأ).

حتى لو كان الاستخدام المزدوج للمتغير واضح بالنسبة إلك، فإنه لن يكون واضح لشخص آخر. سوف يذهلك الوضوح الإضافي الذي تنجزه باستخدام متغيرين لمعالجة نوعين من المعلومات. ولا أحد سوف يحسدك على التخزين الإضافي.

بيانات مثبتة

تأكد من استخدام جميع المتغيرات المُصرّح عنها. إن مُضاد استخدام متغير لأكثر من غرض واحد هو بعدم استخدامه على الإطلاق. وجدت دراسة من قبل كارد، وتشيرتش، وأغريستي، ارتباط المتغيرات غير المستخدمة بمعدلات أعلى من الأخطاء (1986). احصل على عادة التحقق من كونك استخدمت جميع المتغيرات التي تم التصريح عنها. يُعلم بعض المترجمات البرمجية وبعض الوحدات (مثل lint) المتغيرات غير المستخدمة كتحذير.

لائحة اختبار: اعتبارات عامة في استخدام البيانات1

تهيئة المتغيرات2

- هل تتحقق كل إجرائية من صلاحية وسطاء الدخل؟
- هل تقوم الشفرة بالتصريح عن المتغيرات قريباً لمكان استخدامها الأول؟
- هل تقوم الشفرة بتهيئة المتغيرات، حالما يتم التصريح عنها، إذا كانت هناك إمكانية لذلك؟
- هل تقوم الشفرة بتهيئة المتغيرات قريباً من مكان استخدامها الأول، إذا لم تكن هناك إمكانية للتصريح وتهيئتهم في الوقت نفسه؟
- هل تم تعريف العدادات والمراكمات بشكل صحيح، وإذا كان هناك ضرورة، هل تمت إعادة تهيئتهم في كل مرة يتم استخدامهم.
- هل تم إعادة تهيئة المتغيرات بشكل صحيح في الشفرة، التي يتم تنفيذها بشكل متكرر؟
- هل تتم ترجمة الشفرة بدون أية تحذيرات من قبل المترجم البرمجي؟ (وهل قمت بتشغيل كل التحذيرات المتاحة؟)
- إذا كانت لغة البرمجة المُستخدمة تستخدم تصاريح ضمنية، فهل عوضت عن المشاكل التي تسببها؟

قضايا عامة أخرى في استخدام البيانات

- هل لدى كل المتغيرات أصغر نطاق ممكن؟
- هل المراجع للمتغيرات قريبة من بعضها البعض قدر الإمكان، سواء من كل مرجع لمتغير إلى المرجع التالي، وبالمجمل زمن الحياة؟
- هل توافق بنى التحكم أنواع البيانات؟
- هل يتم استخدام كل المتغيرات المُصرح عنها؟
- هل كل المتغيرات مرتبطة عند الأوقات المناسبة- هذا هو، هل تحافظ على توازن مُدهش بين مرونة معدل الارتباط والتعقيد المتزايد المرتبط مع الارتباط في وقت متأخر؟
- هل يملك كل متغير غرض واحد وفقط غرض واحد؟
- هل معنى كل متغير صريح، بدون معاني خفية؟

---

[1] cc2e.com/1092

[2] **إشارة مرجعية**: للمزيد حول قائمة التحقق التي تُطبق على أنواع خاصة من البيانات بدلاً من القضايا العامة، انظر إلى قائمة التحقق في الفصل 12 "أنواع البيانات الأساسية"، الصفحة 460. وللمزيد حول قضايا تسمية المتغيرات، انظر قائمة التحقق في الفصل 11 "قوة أسماء المتغيرات".

النقاط المفتاحية

- إن عملية تهيئة البيانات هي عرضة للأخطاء، لذلك استخدم تقنيات التهيئة الموصوفة في هذا الفصل لتجنب المشاكل المُحدثة بواسطة قيم التهيئة غير المتوقعة.
- صغّر النطاق لكل متغير. حافظ على المراجع لمتغير قريبة من بعضها البعض. حافظ على المتغير ليكون محلي بالنسبة لإجرائية أو صف. تجنب البيانات العامة.
- حافظ على العبارات البرمجية التي تعمل مع نفس المتغيرات قريبة من بعضها البعض قدر الإمكان.
- يميل الارتباط المبكر إلى الحد من المرونة ولكنه يقلل التعقيد. يميل الارتباط المتأخر إلى زيادة المرونة ولكن مع ضريبة زيادة التعقيد.
- استخدم كل متغير من أجل غرض واحد وفقط غرض واحد.

الفصل 11

قوّة أسماء المتحولات

المحتويات[1]

مواضيع ذات صلة

- أسماء الإجرائيات: القسم 7. 3
- أسماء الصفوف: القسم 5. 2
- قضايا عامة في استخدام المتغيرات: الفصل 10
- تنسيق التصريحات عن البيانات: "تخطيط التصريحات عن البيانات" في القسم 31. 5
- توثيق المتغيرات: "التعليق على التصريحات عن البيانات" في القسم 32. 5

بقدر أهمية موضوع الأسماء الجيدة للبرمجة الفعالة، لم أقرأ أبداً نقاشاً غطى أكثر من حفنة دزينات من الاعتبارات التي تتعلق بإنشاء أسماء جيدة. تكرس العديد من الكتب البرمجية عدة مقاطع لاختيار الاختصارات، وتأخذ بقول الأحاديث المبتذلة، وتتوقع منك أن تصون نفسك. انا أنوي أن أكون مجرم بالعكس: أفيض عليك بمعلومات عن الأسماء الجيدة أكثر مما ستستفيد منه للأبد!

[1] cc2e.com/1184

تُطبق الخطوط العريضة لهذا الفصل في تسمية المتحولات-الكائنات والبيانات البدائية. لكنها أيضا تُطبق على تسمية الصفوف والحزم والملفات والكيانات البرمجية الأخرى. لتفاصيل حول تسمية الإجرائيات، راجع القسم 7. 3، "أسماء جيدة للإجرائيات."

11. 1 اعتبارات في اختيار أسماء جيدة

لا تستطيع أن تعطي المتغير اسم بالطريقة التي تعطي فيها كلباً اسم-لأنه ظريف أو لأن مسمعه جيد. بشكل مغاير للكلب واسمه، الذين هما كيانين مختلفين، إن المتحول واسم المتحول هما بالجوهر نفس الشيء. إذن، حسن أو سوء المتحول يتحدد بشكل كبير من اسمه. اختر أسماء المتحولات بعناية. هاك مثال عن شفرة تستخدم أسماء سيئة للمتحولات:



مثال بلغة جافا عن أسماء متحولات رديئة:

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

ما الذي يحدث في هذه القطعة من الشفرة؟ ماذا تعني x1 وxx وxxx؟ ماذا تعني fido؟ افترض أن شخصاً ما أخبرك أن هذه الشفرة تحسب مجموع فاتورة الزبون بالاعتماد على الرصيد الجاري ومجموعة جديدة من المشتريات أي متحول ستستخدم لتطبع فاتورة الزبون للمجموعة الجديدة من المشتريات فقط؟

إليك إصدار من نفس الشفرة يجعل الإجابة على هذه الأسئلة أسهل:

مثال بلغة جافا عن أسماء جيدة للمتحولات:

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax(
newPurchases );
balance = balance + LateFee( customerID, balance )
+ monthlyTotal;
balance = balance + Interest( customerID, balance
);
```

بالنظر الى الفرق بين هاتين القطعتين من الشفرة، اسم المتحول الجيد يكون مقروء وسهل التذكر ومناسب. يمكنك ان تستخدم عدة قواعد تعتمد على الخبرة لإنجاز هذه الأهداف.

أهم اعتبار في التسمية



أهم اعتبار في تسمية متحول هو أن يصف الاسم بدقة وكمالٍ الكيان الذي يمثله المتحول. إليك تقنية فعالة للحصول على اسم جيد وهي أن تذكر بالكلمات ما يمثله المتحول. غالبا هذه العبارة ذاتها هي الاسم الأفضل للمتحول. إنها سهلة القراءة لأنها لا تحوي اختصارات مشفرة، وهي واضحة. ولأنها وصف كامل للكيان، لن تكون مربكة مع الأشياء الأخرى. وهي سهلة التذكر لأن الاسم مشابه للمفهوم.

لمتحول يمثل عدد الأشخاص في فريق الولايات المتحدة الأولمبي، ستعطي اسماً مثل numberOfPeopleOnTheUsOlympicTeam. المتحول الذي يمثل عدد المقاعد في الصالة سيكون numberOfSeatsInTheStadium. المتحول الذي يمثل العدد الأقصى للنقاط المحققة من قبل الفريق الوطني في أولمبياد حديثة سيكون maximumNumberOfPointsInModernOlympics. المتحول الذي يحوي معدل الفائدة الحالية من الأفضل أن يسمى rate أو interestRate بدلاً من r أو x. فهمتَ الفكرة.

لاحظ مميزتين لهذه الأسماء. أولاً، إنها سهلة في فك التشفير. في الحقيقة، إنها لا تحتاج أن يتم فك تشفيرها على الإطلاق لأنك تستطيع ببساطة أن تقرأها. لكن ثانياً، بعض الأسماء طويل-طويل جداً ليكون عملياً. سأصل إلى السؤال عن طول اسم المتحول قريباً.

الجدول 11-1 يظهر عدة أمثلة عن أسماء متحولات، جيدة وسيئة:

جدول 11-1 أمثلة عن أسماء متحولات جيدة وسيئة

الغرض من المتحول	أسماء جيدة، معرّفات جيدة	أسماء سيئة، معرفات رديئة
المجموع الجاري للشيكات المكتوبة للتاريخ	checkTotal ،runningTotal	x ،CHKTTL ،checks ،ct ،written، x2 ،x1
سرعة القطار السريع	،trainVelocity ،velocity velocityInMph	train ،x2 ،x1 ،x ،tv ،v ،velt
التاريخ الحالي	today's Date ،current Date	date ،x2 ،x1 ،x ،c ،current ،cd
الأسطر في الصفحة	linesPerPage	x2،x1 ،x ،l ،lines ،lpp

الاسمين currentDate، todaysDate هما اسمان جيدان لأنهما يصفان بدقة وكمال فكرة "التاريخ الحالي" "currentDate." بالحقيقة، إنهما يستخدمان كلمات جليّة. يقوم المبرمجون أحياناً باستخدام الكلمات الاعتيادية، والتي تكون غالباً الحل الأسهل. لأنهما قصيران جداً وليسا على الإطلاق معبران، c ،cd اسمان رديئان. current رديء لأنه لا يخبرنا ما هو الحالي. date هو تقريباً اسم جيد، لكنه اسم رديء بالتحليل النهائي لأن التاريخ المضمن ليس مجرد أي تاريخ، بل التاريخ الحالي؛ date بذاتها لا تعطي هكذا إيحاء. x، x1، x2 هي أسماء رديئة لأنها أسماء رديئة دوما-x بشكل تقليدي تمثل قيمة مجهولة؛ إذا لم تريد أن تكون متغيراتك قيم مجهولة، فكر بأسماء أفضل.



ينبغي أن تكون الأسماء محددة قدر الإمكان. أسماء مثل x، temp، i، والتي هي عامة بما فيه الكفاية حتى يتم استخدامها لأكثر من غرض واحد، ليست مرشدة بقدر يمكن أن يكون وهي عادة أسماء سيئة.

التوجه إلى المشكلة

الاسم الجيد القابل للتذكر بشكل عام يعطي معلومة عن المشكلة بدلاً من الحل. الاسم الجيد يميل إلى التعبير عن ماذا وليس كيف. بالعموم، إذا أشار الاسم إلى بعض المفاهيم الحاسوبية بدلاً من الإشارة إلى المشكلة، إنها كيف وليست ماذا. تجنب هكذا اسم من أجل اسم يشير إلى المشكلة بذاتها.

يمكن أن يُدعى سجل لبيانات الموظفين inputRec أو inputRec .employeeData هو مصطلح حاسوبي يشير إلى أفكار حاسوبية-إدخال وسجل. employeeData يدل على مجال المشكلة وليس على عالم الحوسبة. بشكل مشابه، من أجل حقل مكون من بت يحدد حالة الطابعة، bitFlag هو اسم أكثر حاسوبية من printerReady. في تطبيق محاسبة، calcVal هو حاسوبي أكثر من sum.

طول الاسم المثالي

يبدو أن الطول المثالي للاسم موجود في مكان ما بين طول x وmaximumNumberOfPointsInModernOlympics. لا تعطي الأسماء القصيرة جداً معنىً كاف. المشكلة بأسماء مثل x1، x2 هي المشكلة التي لا تعرف فيها أي شيء عن العلاقة بين x1 وx2 حتى لو قدرت أن تكتشف ما هي x.

الأسماء الطويلة جداً هي صعبة الكتابة ويمكن أن تخربط البنيان المرئي للبرنامج.

بيانات مثبتة

وجد غورلا، بيناندر، وبيناندر أنه تم تقليل الجهد المطلوب لتصحيح برنامج عندما كان متوسط طول المتحولات بين 10 إلى 16 محرف (1990). كانت البرامج التي لديها متوسط طول أسماء من 8 إلى 20 محرف تقريباً بنفس درجة السهولة للتصحيح. لا يعني هذا التوجيه أنه عليك أن تحاول أن تجعل كل أسماء متغيراتك بطول بين 9 و15 أو 10 و16 محرفاً. إنه يعني بالتأكيد أنه إذا ألقيت نظرة إلى شفرتك ورأيت عدة أسماء أقصر، فعليك أن تتأكد أن هذه الأسماء واضحة بقدر الحاجة.

ربما ستخرج مباشرة باتخاذ طريقة "غولديلوكس والدبة الثلاثة"[1] لتسمية المتحولات، كما يوضح الجدول 11-2.

جدول 11-2 أسماء متغيرات طويلة جداً أو قصيرة جداً أو على القد

طويل جداً	numberOfPeopleOnTheUsOlympicTeam numberOfSeatsInTheStadium maximumNumberOfPointsInModernOlympics
قصير جداً	n، np، ntm

[1] غولديلوكس والدبة الثلاثة قصة للأطفال Goldilocks and the Three Bears

nsisd ،ns ،n points ،max ،mp ،m	
teamMemberCount ،numTeamMembers seatCount ،numSeatsInStadium pointsRecord ،teamPointsMax	مناسب

تأثير المجال على أسماء المتغيرات(المتحولات)

هل أسماء المتحولات القصيرة دائماً سيئة؟ لا، ليس دائماً.1 عندما تعطي متحول اسماً قصيراً مثل i، فإن الطول نفسه يخبرك شيئاً ما عن المتحول-أعني، أن المتحول تم إنشاؤه على عجل لمجال محدود من العمليات.

عندما يقرأ المبرمج متحول كهذا ينبغي أن يكون قادراً على افتراض أن قيمته لا تستخدم بعد عدّة أسطر من الشفرة. عندما تسمي متحول i، فإنك تقول، "هذا المتحول دليل مصفوفة أو عداد حلقة عادي وليس له أي معنى خارج هذه الاسطر القليلة من الشفرة."

وجدت دراسة قام بها هانسن (W. J. Hansen) أن الأسماء الأطول أفضل للمتحولات القليلة الاستخدام أو المتحولات الشاملة، والأسماء الأقصر أفضل للمتحولات المحلية أو متحولات الحلقات (شنيديرمان 1980) (Shneiderman 1980). الأسماء القصيرة هي موضوع للعديد من المشاكل، على أية حال، بعض المبرمجين الحذرين يتجنبونها بشكل كامل كإحدى سياسات البرمجة الوقائية.

استخدم المعدلات على الأسماء الموجودة في فضاء الأسماء الشامل2 إذا كان لديك متحولات في فضاء الأسماء الشامل (الثوابت المسماة وأسماء صفوف الخ)، ضع في حسبانك أنه من الممكن أن تحتاج أن تتبنى عرف لتجزئة فضاء الأسماء الشامل وتجنب التعارض في التسميات. في سي++ وسي# بإمكانك استخدام الكلمة المفتاحية namespace لتجزئ فضاء الأسماء الشامل.

مثال بلغة سي++ لاستخدام الكلمة المفتاحية namespace لتجزيء فضاء الأسماء الشامل

```
namespace UserInterfaceSubsystem {
...
// lots of declarations
...
}
namespace DatabaseSubsystem {
...
// lots of declarations
...
}
```

[1] **إشارة مرجعية** نوقش المجال بتفصيل أكثر في القسم 10. 4 "المجال".

[2] المقصود ب المعدلات هنا: كلمات تدخل على كلمات أخرى فتضيف أو تغير في معناها

إذا صرحت عن صف Employee في كلا UserInterfaceSubsystem وDatabaseSubsystem، بإمكانك أن تحدد أي واحد تريد أن تشير إليه بكتابة UserInterfaceSubsystem::Empoyee أو DatabaseSubsystem::Employee. في جافا، يمكنك إنجاز نفس الشيء باستخدام الحزم(packages).

لا تزال تستطيع أن تستخدم أعراف التسمية في اللغات التي لا تدعم فضاء الأسماء أو الحزم. أحد الأعراف أن تطلب لزوماً أن توضع بادئة تساعد على تذكر النظام الفرعي أمام أسماء الصفوف المرئية بشكل شامل. يمكن أن يصبح صف الموظف التابع لواجهة المستخدم uiEmployee (user interface)، ويمكن أن يصبح صف الموظف التابع لقاعدة البيانات dbEmplolyee، هذا يقلل من خطر التصادمات في فضاء الأسماء الشامل.

معدلات "القيم المحسوبة" في أسماء المتحولات

تحتوي العديد من البرامج على متحولات تحوي قيم محسوبة: المجاميع والقيم العليا والمتوسطات وهلم جراً. إذا عدلت اسماً بمعدل من المجموعة: Total، Sum، Average، Max، Min، Record، String، Pointer ضع المعدل في نهاية الاسم.

يقدم هذا التطبيق عدة حسنات. أولاً، القسم الأكثر أهمية من اسم المتحول، القسم الذي يعطي المتحول معظم معناه، موجود في البداية، لذا فإنه بارز إلى أقصى حد وتتم قراءته أولاً. ثانياً، بمباشرة هذا العرف، إنك تتجنب الارتباك الذي يمكن أن تخلقه إذا كنت تستخدم كلا totalRevenue وrevenueTotal في نفس البرنامج. الاسمان متساويان دلالياً، والعرف سيمنع استخدامها كما لو كانا مختلفين. ثالثاً، مجموعة الأسماء مثل revenueTotal، expenseTotal، revenueAverage، expenseAverage تمتلك تماثلاً مُرْضياً. مجموعة أسماء مثل totalRevenue، expenseTotal، revenueAverage، averageExpense لا تؤدي إلى الشعور بالترتيب. أخيراً، إن التماسك يعزز إمكانية القراءة ويسهل الصيانة.

الموضع الاختياري للمعدل Num هو استثناء لقاعدة القيم المحسوبة توضع في نهاية الاسم. بوضعه في البداية، Num يشير إلى المجموع: numCustomers هو مجموع الزبائن. بوضعه في النهاية، Num يشير إلى الدليل: customerNum هو رقم الزبون الحالي. حرف s في نهاية numCustomers هو معلومة مخبأة تدل على الفرق في المعنى. لكن لأن استخدام Num كثيراً ما يخلق ارتباكاً، ربما من الأفضل أن "نخطو بجانب" القضية كلها باستخدام Count أو Total للإشارة إلى مجموع الزبائن وIndex للإشارة إلى الزبون المحدد. هكذا إذن، customerCount هو مجموع الزبائن وcustomerIndex يشير إلى زبون محدد.

تضادات شائعة في أسماء المتحولات

استخدم التضادات بدقة.[1] استخدام أعراف التسمية للتضادات يساعد على التماسك، والذي يساعد على إمكانية القراءة. أزواج مثل begin/end سهلة الفهم والتذكر. الأزواج التي تحيد عن تضادات "اللغة الشائعة" تميل لتكون صعبة التذكر ولذا تكون مربكة. إليك بعض التضادات الشائعة:

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

11. 2 تسمية أنواع محددة من البيانات

بالإضافة إلى الاعتبارات العامة في تسمية البيانات، تظهر اعتبارات خاصة في تسمية أنواع محددة من البيانات. هذا القسم يوضح الاعتبارات المتخصصة لمتحولات الحلقات ومتحولات الحالة والمتحولات المؤقتة والمتحولات المنطقية والأنماط التعدادية والثوابت المسماة.

تسمية فهارس الحلقات

لقد ظهرت التوجيهات بخصوص تسمية المتحولات في الحلقات لأن الحلقات وظيفة شائعة بكثرة في برمجة الحاسوب.[2] الأسماء i، j، k متعارف عليها:

مثال بلغة جافا عن اسم متحول حلقة بسيطة

[1] **إشارة مرجعية** للائحة مشابهة عن التضادات في أسماء الإجرائيات، راجع "استخدم التضادات بدقة" في القسم 7. 3.

[2] **إشارة مرجعية** لتفاصيل حول الحلقات، راجع الفصل 16 "التحكم بالحلقات."

```
for ( i = firstItem; i < lastItem; i++ ) {
data[ i ] = 0;
}
```

إذا أريد من المتحول أن يستخدم خارج الحلقة، ينبغي أن يعطى اسماً معبراً أكثر من i، j، k مثالاً. إذا كنت تقرأ من ملف وتريد أن تتذكر كم سجل قرأت، اسم مثل recordCount سيكون مناسباً:

مثال بلغة جافا عن اسم واصف وجيد لمتحول حلقة

```
recordCount = 0;
while ( moreScores() ) {
score[ recordCount ] = GetNextScore();
recordCount++;
}
// lines using recordCount
...
```

إذا كانت الحلقة أطول من عدة أسطر، فمن السهل أن تنسى ما المفروض أن يمثله i وإنه من الأفضل إعطاء فهرس الحلقة اسماً معبراً أكثر. لأن الشفرة غالبا ما تتغير وتتوسع وتنسخ إلى برامج أخرى، فالعديد من المبرمجين ذوي الخبرة يتجنبون أسماء مثل i بشكل كامل.

أحد الأسباب الشائعة لتوسع الحلقات هو أن تعشّش. عين أسماء أطول لمتحولات الحلقات لتحسن إمكانية القراءة.

مثال بلغة جافا عن أسماء حلقات جيدة في الحلقات المعششة

```
for ( teamIndex = 0; teamIndex < teamCount;
teamIndex++ ) {
    for ( eventIndex = 0; eventIndex < eventCount[
teamIndex ]; eventIndex++ ) {
        score[ teamIndex ][ eventIndex ] = 0;
    }
}
```

الأسماء المنتقاة بعناية لمتحولات فهارس الحلقات تمنع المشاكل الشائعة لتبادل الكلام المتعلق بالفهارس: كقولك i عندما تعني j و j عندما تعني i. كما إنها تجعل الدخول إلى المصفوفة أوضح: score[teamIndex][eventIndex] أكثر دلالة من score[i] [j].

إذا توجب عليك استخدام i، j، k، فلا تستخدمها في أي شيء إلا فهارس الحلقات وللحلقات البسيطة-هذا العرف أيضاً معترف به جيداً، وكسره باستخدام تلك الأسماء بطرق أخرى مربك. الطريقة الأبسط لتجنب هكذا مشاكل ببساطة هي التفكير بأسماء أكثر وصفاً من i، j، k.

تسمية متحولات الحالة

متحولات الحالة تصف حالة البرنامج. هاك توجيه التسمية:

فكر باسم أفضل من علم من أجل متحولات الحالة من الأفضل التفكير بالأعلام كمتحولات حالة. العلم ينبغي ألا يحوي أبداً flag في اسمه لأنه لا يعطيك أي فكرة عم يفعل العلم. للتوضيح، ينبغي للأعلام أن تكون قيم مسندة وقيمها ينبغي أن تُختبر بالأنماط التعدادية أو الثوابت المسماة أو المتحولات الشاملة التي تلعب دور الثوابت المسماة. إليك بعض الأمثلة عن الأعلام بأسماء سيئة:

مثال بلغة سي++ عن الأعلام المشفرة




```
if ( flag ) ...
if ( statusFlag & 0x0F ) ...
if ( printFlag == 16 ) ...
if ( computeFlag == 0 ) ...
flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

عبارات مثل statusFlag = 0x80 لا تعطيك أي دليل عم تفعل الشفرة إلا إذا كنت من كتب الشفرة أو كان لديك توثيق يخبرك ما يمثل statusFlag وما تمثل x800. هنا مثال عن شفرة مكافئة أوضح:

مثال بلغة سي++ عن استخدام أفضل لمتحولات الحالة

```
if ( dataReady ) ...
if ( characterType & PRINTABLE_CHAR ) ...
if ( reportType == ReportType_Annual ) ...
if ( recalcNeeded = false ) ...
dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

بشكل أوضح، characterType = CONTROL_CHARACTER أكثر دلالة من statusFlag = 0x80، كذلك، (العبارة) الشرطية if (reportType == ReportType_Annual) أوضح من if (printFlag == 16). المثال الثاني يظهر أنك تستطيع استخدام هذا النهج مع الأنماط التعدادية بنفس الجودة مع الثوابت المسماة المعرفة مسبقاً. إليك كيف تستطيع استخدام الثوابت المسماة والأنماط التعدادية لتهيئ القيم المستخدمة في المثال:

التصريح عن متحولات الحالة في لغة سي++

```
// values for CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x02;
const int PUNCTUATION = 0x04;
const int LINE_DRAW = 0x08;
const int PRINTABLE_CHAR = ( LETTER | DIGIT |
PUNCTUATION | LINE_DRAW );
const int CONTROL_CHARACTER = 0x80;
// values for ReportType
enum ReportType {
ReportType_Daily,
ReportType_Monthly,
ReportType_Quarterly,
ReportType_Annual,
ReportType_All
};
```

عندما تجد نفسك تكتشف قسماً من الشفرة، ضع في حسبانك أن تعيد تسمية المتحولات. من الجيد أن تكتشف غوامض الجريمة، لكن لا ينبغي أن تحتاج اكتشاف الشفرة. ينبغي أن تكون قادراً على قراءتها.

تسمية المتحولات المؤقتة

يُستخدم المتحول المؤقت ليحمل النتيجة الوسيطة للحسابات، كقائم مقام مؤقت، وليحمل قيمة متعلقة بتدبير الشؤون الداخلية للبرنامج. عادة ما تسمى المتحولات المؤقتة temp أو x أو أي اسم غامض غير معبر. بالعموم، المتحولات المؤقتة هي إشارة أن المبرمج لما يفهم تماماً المشكلة. أكثر بعد، لأن المتحولات بشكل رسمي لها حالة "مؤقتة"، فإن المبرمجين يميلون للتعامل معها بشكل أكثر عفوية من باقي المتحولات، مما يزيد احتمال الأخطاء.

كن حذراً من المتحولات "المؤقتة" غالباً ما يكون ضروريا أن تحفظ قيماً بشكل مؤقت. لكن بطريقة أو أخرى، الغالبية العظمى من المتحولات في برنامجك مؤقتة. تسمية مجموعة منها أنها مؤقتة ربما يدل على أنك لست متأكد من الغرض الحقيقي لها. ضع في حسبانك الأمثلة التالية:

مثال بلغة C++ عن أسماء متحولات "مؤقتة" غير معبرة

```
// Compute solutions of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
temp = sqrt( b^2 - 4*a*c );
solution[0] = ( -b + temp ) / ( 2 * a );
solution[1] = ( -b - temp ) / ( 2 * a );
```

إنه لحسن أن تخزن قيمة التعبير `sqrt( b^2 – 4 * a * c )` في متحول، وخصوصاً أنها استخدمت في مكانين لاحقاً. لكن الاسم temp لا يخبرنا أي شيء عم يفعل المتحول. في المثال التالي نهج أفضل:

مثال بلغة سي++ أبدل فيه اسم متحول "مؤقت" بمتحول حقيقي

```
// Compute solutions of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
discriminant = sqrt( b^2 - 4*a*c );
solution[0] = ( -b + discriminant ) / ( 2 * a );
solution[1] = ( -b - discriminant ) / ( 2 * a );
```

هذا بالجوهر نفس الشفرة، لكنها محسَّنة باستخدام اسم متحول دقيق ومعبر.

تسمية المتحولات المنطقية

فيما يلي عدة توجيهات للتطبيق على تسمية المتحولات المنطقية:

حافظ على الأسماء المنطقية القياسية في فكرك هنا بعض أسماء المتحولات المنطقية المفيدة بشكل خاص:

- **done** ا ستخدمه لتدل إن تم عمل شيء ما. يمكن أن يدل المتحول إن تم عمل الحلقة أو تم عمل بعض الأشياء الأخرى. اجعل done بقيمة false قبل أن يتم عمل شيء ما، واجعله true عند إكمال ذلك الشيء.
- **error** اســتـخدمه لتدل على حدوث خطأ ما اجعل المتحول false عندما لا يكون قد حدث خطأ وtrue عندما يكون قد حدث خطأ
- **found** اســـتخدمه لتدل إن تم إيجاد قيمة ما، اجعل المتحول false عندما لا تكون قد وُجدت القيمة وtrue حالما توجد القيمة. اســتخدم found عندما تبحث في مصــفوفة عن قيمة، وفي ملف عن معرف موظف، وفي لائحة شيكات المدخول عن مقدار محدد للمدخول، وإلى ما هنالك.
- **success** أو ok اســتخدمه لتدل إن نجحت عملية. اجعل المتحول false عندما تكون العملية قد فشــلت وtrue عندما تكون العملية قد نجحت. إذا اســتطعت، اســتبدل success باســم أكثر تمييز يصــف بدقة ما الذي وقع عليه النجاح. إذا كان البرنامج يوصــف بأنه ناجح عند إكمال المعالجة، بإمكانك أن تســتخدم proccessingComplete بدلاً. إذا كان البرنامج يوصــف بأنه ناجح عند إيجاد قيمة، بإمكانك أن تســتخدم found بدلاً.

أعط المتحولات المنطقية أسماء توحي ب صح أو خطأ (`true or false`) أسماء مثل done وsuccess هي أسماء منطقية جيدة لأن الحالة إما صح أو خطأ (`true or false`)، شيء ما تم القيام به أو لم؛ إنه نجاح أو ليس. أسماء مثل `status` و`sourceFile`، بالجهة الأخرى، هي أسماء منطقية رديئة لأنها ليست بشكل واضح صح أو خطأ. ما الذي يعينه أن تكون الحالة صح؟ هل تعني أن شيء ما يمتلك حالة؟ كل شيء له حالة. هل صح

يعني أن حالة شيء ما تمام؟ أو هل خطأ تعني أن لا شيء جرى بطريقة خطأ؟ باسم مثل status، لا يمكنك أن تخبرنا.

من أجل نتائج أفضل، استبدل status باسم مثل error أو statusOK، واستبدل sourceFile ب sourceFileAvailable أو sourceFileFound، أو أي شيء يمثله المتحول.

بعض المبرمجين يحبون أن يضعوا is في بداية الأسماء المنطقية. وعندها يصبح اسم المتحول سؤال: isdone? isError? isFound? isProccessingComplete؟ الجواب على السؤال بصح أو خطأ يقدم قيمة المتحول. إحدى فوائد هذا النهج أنه لا ينجح مع الأسماء الغامضة: isStatus؟ لا تعطي أي معنى على الإطلاق. إحدى السلبيات أنه يجعل التعابير المنطقية البسيطة أقل قابلية للقراءة: (if (isFound بشكل طفيف أقل قابلية للقراءة من (if (found.

استخدم اسماء إيجابية للمتحولات المنطقية الأسماء السلبية مثل notFound، notdone، notSuccessful صعبة القراءة عندما تُنفى-على سبيل المثال،

```
if not notFound
```

اسم كهذا يجب أن يُستبدل ب found أو done أو processingComplete وبعدها يُنفى بمعامل حسب المناسب. إن وُجد ما تبحث عنه، فلديك found بدلاً من not notFound.

تسمية الأنماط التعدادية

عندما تستخدم الأنماط التعدادية، تستطيع أن تضمن وضوح انتماء عناصر النوع إلى نفس المجموعة باستخدام بادئة المجموعة،[1] مثل _Color، _Planet، _Month. هنا بعض الأمثلة عن تعريف عناصر الأنواع التعدادية باستخدام البوادئ:

مثال بلغة فيجوال بيسك عن استخدام عرف تسمية البوادئ للأنماط التعدادية

[1] **إشارة مرجعية** لتفاصيل حول استخدام الأنماط التعدادية، انظر القسم 12. 6، "الأنماط التعدادية"

```
Public Enum Color
Color_Red
Color_Green
Color_Blue
End Enum
Public Enum Planet
Planet_Earth
Planet_Mars
Planet_Venus
End Enum
Public Enum Month
Month_January
Month_February
...
Month_December
End Enum
```

بالإضافة، فإن النمط التعدادي نفسه (Month ،Planet ،Color) يمكن أن تُعرّف بعدة أشكال، منها الكل بأحرف كبيرة أو باستخدام البوادئ (e_Month ،e_Planet ،e_Color) يمكن أن يجادل شخص ما أنّ النمط التعدادي هو جوهرياً نمط "معرف من المستخدم" ولذا اسم النمط التعدادي ينبغي أن يُنسق بالطريقة التي تنسق فيها الأنماط "المعرفة من المستخدم" الأخرى مثل الصفوف. وجدال آخر مختلف أن الأنماط التعدادية هي أنماط، لكنها أيضاً ثوابت، لذا ينبغي أن تُنسق بالطريقة التي تنسق بها الثوابت. يستخدم هذا الكتاب عرف لحالة مركبة بخصوص أسماء الأنماط التعدادية.

في بعض اللغات، تُعامل الأنماط التعدادية بشكل أقرب إلى معاملة الصفوف، وعناصر التعدادية تكون دائما مسبوقة باسم النمط التعدادي، مثل Color.Color_Red، Planet.Planet_Earth. إذا كنت تعمل على ذلك النوع من اللغات، فإن تكرار البادئة يعطي معنى أقل، لذا بإمكانك أن تعامل اسم النمط التعدادي نفسه على أنه بادئة وتبسط الأسماء إلى Color.Red وPlanet.Earth.

تسمية الثوابت

عند تسمية الثوابت، اعطِ اسم للكيان المجرد الذي يمثله الثابت بدلاً من الرقم الذي يشير إليه.[1] FIVE اسم سيء لثابت (بغض النظر إن كانت القيمة الممثلة هي 5.0). CYCLES_NEEDED اسم جيد. CYCLES_NEEDED يمكن أن يساوي 5.0 أو 6.0. FIVE = 6.0 سيكون مضحكاً. على نفس النموذج، BAKERS_DOZEN اسم رديء لثابت، DONUTS_MAX اسم جيد لثابت.

[1] **إشارة مرجعية** لتفاصيل حول استخدام الثوابت المسماة، انظر القسم 12. 7، "الثوابت المسماة."

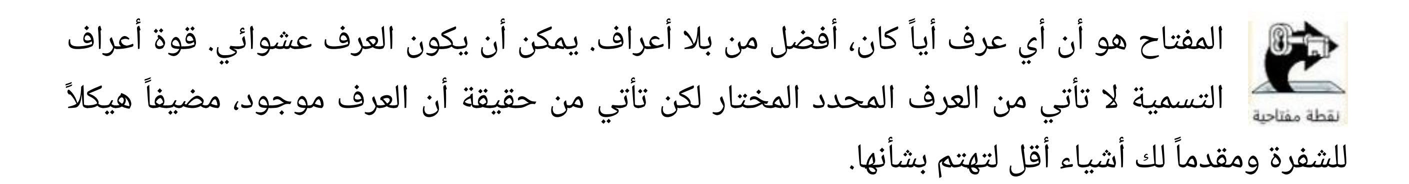
11.3 قوة أعراف التسمية

يقاوم بعض المبرمجين المعايير والأعراف-ولأسباب جيدة. بعض المعايير والأعراف تكون جامدة وغير فعالة-هادمة للإبداع وجودة البرنامج. إن هذا غير سار منذ أن المعايير الفعالة هي بعضٌ من الأدوات الأكثر قوة التي بيمينك. هذا القسم يناقش لم ومتى وكيف ينبغي أن تنشئ معاييرك الخاصة لتسمية المتحولات.

لم نريد الأعراف؟

الأعراف تؤمن عدة منافع محددة:

- إنها تدعك تأخذ الأمور على أنها مضــمونة. باتخاذ قرار واحد شـــامل بدلاً من عدة قرارات محلية، بإمكانك ان تركز على المميزات الأكثر أهمية في الشفرة
- إنها تساعدك في نقل المعرفة عبر المشـاريع. التشـابهات في الأسـماء تعطيك فهم أسـهل وموثوق أكثر عم يفترض للمتحولات غير المألوفة أن تعمل.
- إنها تســاعدك في تعلم الشــفرة بســرعة أكبر في المشــاريع الجديدة. بدلاً من تعلم أن شــفرة أنيتا تبدو هكذا، وجوليا هكذا، وكريستين يحب شيء آخر، بإمكانك أن تعمل بمجموعة أكثر تماسك من الشفرة.
- إنها تخفض تكاثر الأســماء. بدون أعراف التســمية، يمكنك ببســاطة أن تدعو شــيء واحد باســمين مختلفين. على ســبيل المثال، بإمكانك أن تدعو مجموع النقاط بكلا pointTotal وtotalPoints. ربما لا يكون هذا مربكاً لك عند كتابة الشـفرة، لكنه يمكن أن يكون مربكاً كثيراً بالنسـبة لمبرمج جديد يقرؤها لاحقاً.
- إنها حيوية لنقاط ضــعف اللغة. يمكنك أن تســتخدم الأعراف لتحاكي الثوابت المســماة والأنماط التعدادية. الأعراف يمكن أن تتباين بين محلية وتابعة للصف وبيانات شاملة ويمكن أن تدمج معلومات النمط من أجل أنماط غير مدعومة من المترجم.
- إنها تؤكد العلاقات بين العناصــر المترابطة. إذا كنت تســتخدم بيانات كائن، يهتم المترجم بهذا تلقائياً، إذا كانت لغتك لا تدعم الكائنات، بإمكانك تزويدها بعرف تسمية. أسماء مثل address وphone وname لا تدل أن المتحولات مترابطة. لكن افترض أنك قررت بأن على كل متحولات "بيانات الموظف" أن تبدأ ببادئة Employee. EmployeeAddres، employeePhone، employeeName تأخذ كل الشـكوك في ترابط هذه المتحولات. يمكن للأعراف البرمجية أن تعوض عن ضعف اللغة التي تستخدمها.



المفتاح هو أن أي عرف أياً كان، أفضل من بلا أعراف. يمكن أن يكون العرف عشوائي. قوة أعراف التسمية لا تأتي من العرف المحدد المختار لكن تأتي من حقيقة أن العرف موجود، مضيفاً هيكلاً للشفرة ومقدماً لك أشياء أقل لتهتم بشأنها.

متى ينبغي أن يكون لديك عرف تسمية

لا يوجد قواعد "سريعة وصلبة" حول متى ينبغي أن تتبنى عرف تسمية. لكن هنا عدة حالات فيها تكون الأعراف تستحق التعب من أجلها:

- عندما يكون عدة مبرمجين يعملون على مشروع
- عندما تخطط أن تسلم البرنامج لمبرمج آخر من أجل التعديلات والصيانة (والتي تقريبا تكون دائما)
- عندما تتم مراجعة برامجك من قبل مبرمجين آخرين في منظمتك
- عندما يكون البرنامج كبير جداً بحيث لا تتمكن من الاحتفاظ بكامل الشـيء في دماغك في نفس اللحظة ويجب عليك بالتأكيد أن تفكر به أجزاءً
- عندما يكون البرنامج ذو عمر طويل كفاية، حيث قد تضعه جانبا لعدة أسابيع أو شهور قبل أن تعاود العمل فيه
- عندما يكون لديك العديد من المصطلحات غير العادية التي تكون شائعة في مشروع وتريد أن يكون لديك مصطلحات أو اختصارات معيارية لتستخدمها في كتابة الشفرة

ستستفيد دائماً عندما يكون لديك نوع ما من أعراف التسمية. الاعتبارات السابقة ينبغي أن تساعدك في تحديد حجم العرف كي تستخدمه في مشروع محدد.

درجات الرسمية

الأعراف المختلفة لديها درجات مختلفة من الرسمية.[1] يمكن ان يكون العرف غير الرسمي ببساطة "استخدم أسماء ذات معنى." وصِفت الأعراف غير الرسمية الأخرى في القسم التالي. بالعموم، درجة الرسمية التي تحتاجها تعتمد على عدد الناس المشاركين في البرنامج، وحجم البرنامج، ودورة حياة البرنامج المتوقعة. في المشاريع الصغيرة المبعثرة، يمكن أن يكون العرف الشديد عبئ زائد. في مشاريع أكبر يكون فيها عدة أشخاص، إما من البداية أو خلال دورة حياة البرنامج، تكون الأعراف الرسمية مساعد حيوي للحصول على إمكانية القراءة.

11.4 أعراف التسمية غير الرسمية

تستخدم معظم المشاريع أعراف تسمية نسبياً غير رسمية مثل الموضوعة في هذا القسم.

[1] **إشارة مرجعية** لتفاصيل حول الفوارق في الرسمية في المشاريع الصغيرة والكبيرة، انظر الفصل 27، "كيف يؤثر حجم البرنامج على البناء."

توجيهات للأعراف "المستقلة عن اللغة"

هنا بعض التوجيهات لإنشاء أعراف مستقلة عن اللغة:

باين بين أسماء المتحولات وأسماء الإجرائيات العرف الذي يتبعه هذا الكتاب أن تبدأ أسماء المتحولات والكائنات بحرف صغير وأسماء الإجرائيات بحرف كبير: variableName مقابل RoutineName().

باين بين الصفوف والكائنات التشابه بين أسماء الصفوف وأسماء الكائنات-أو بين الأنماط ومتحولات هذه الأنماط-يمكن أن يكون مخادع. توجد عدة خيارات معيارية، كما هو موضح في الأمثلة التالية

الخيار 1: باين بين الأنماط والمتحولات بتكبير الحرف الأول

```
Widget widget;
LongerWidget longerWidget;
```

الخيار 2: باين بين الأنماط والمتحولات بتكبير كل الأحرف

```
WIDGET widget;
LONGERWIDGET longerWidget;
```

الخيار 3: باين بين الأنماط والمتحولات ببادئة "t_" للأنماط

```
t_Widget Widget;
t_LongerWidget LongerWidget;
```

الخيار 4: باين بين الأنماط والمتحولات ببادئة "a" للمتحولات

```
Widget aWidget;
LongerWidget aLongerWidget;
```

الخيار 5: باين بين الأنماط والمتحولات باستخدام أسماء أكثر تحديداً للمتحولات

```
Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

لكل من هذه الخيارات نقاط قوة وضعف. الخيار 1 هو عرف شائع في اللغات الحساسة لحالة الأحرف بما فيها سي++ وجافا، لكن بعض المبرمجين لا يرتاحون بمباينة الأسماء على قاعدة تكبير الأحرف فقط. بالحقيقة، إنشاء أسماء تختلف فقط بتكبير الحرف الأول من الاسم يبدو أنه يقدم "مسافة نفسية" قصيرة جداً وتمييز مرئي صغير جداً بين الاسمين.

لا يمكن تطبيق نهج الخيار 1 بتماسك في بيئات اللغات المختلطة إذا كانت أي من هذه اللغات غير حساسة لحالة الأحرف. في مايكروسوفت فيجوال بيسك، على سبيل المثال، Dim widget as Widget سيولد خطأ قواعدي لأنه يتم التعامل مع widget وWidget على أنهما نفس الرمز.

يصنع الخيار 2 فرقاً أوضح بين اسم النمط واسم المتحول. لأسباب تاريخية، استُخدمت الأحرف الكبيرة لتحدد الثوابت في سي++ وجافا، على أية حال، وهذا النهج مُعرّض لنفس المشاكل المتعلقة ببيئات اللغات المختلطة والتي يعاني منها الخيار 1.

يعمل الخيار 3 بشكل مناسب في كل اللغات، لكن بعض المبرمجين لا يحبون فكرة البوادئ لأسباب جمالية.

يستخدم الخيار 4 كبديل للخيار 3، لكن لديه سلبية تبديل كل نسخ الصف (الكائنات) بدلاً من تبديل واحد لاسم الصف.

يتطلب الخيار 5 تفكير أكثر بقاعدة كل متحول على حدى. في معظم الحالات، أن تكون مجبراً بالتفكير باسم أكثر تحديداً للمتحول ينتج شفرة مقروءة أكثر. لكن بعض الأحيان أداة (widget) تكون تماماً أداة عامة (generic widget)، وفي هذه الحالات ستجد نفسك حصلت على "دون الأسماء الواضحة"، مثل genericWidget، والتي هي باحتمال ممكن أقل قابلية للقراءة.

باختصار، كل من هذه الخيارات لديها إيجابيات وسلبيات. يستخدم هذا الكتاب شفرة معتمدة على الخيار 5 لأنها الأكثر قابلية للفهم في الحالات التي يكون فيها القارئ غير متآلفاً بالضرورة مع أعراف تسمية أقل بداهة.

ميز المتحولات الشاملة أحد مشاكل البرمجة الشائعة هي سوء استخدام المتحولات الشاملة. إذا أعطيت كل أسماء المتحولات الشاملة بادئة "g_"، على سبيل المثال، المبرمج الذي يرى المتحول g_RunningTotal سيعلم أنه متحول شامل ويتعامل معه على هذا الأساس.

ميز المتحولات الأعضاء عرف البيانات الأعضاء للصف. اجعلها واضحة حيث لا يكون العضو متحول محلي ولا يكون متحول شامل أيضاً. على سبيل المثال، بإمكانك تعريف المتحولات الأعضاء في الصف ببادئة m_ لتحدد أنها بيانات أعضاء.

ميز تعريفات الأنماط أعراف التسمية للأنماط تخدم بشيئين: إنها بوضوح تعرف اسم على أنه اسم نمط، وتمنع التصادمات مع المتحولات. لتطبق هذه الاعتبارات، استخدام بادئة أو لاحقة نهج جيد. في سي++، النهج الشائع هو استخدام كل الأحرف كبيرة لاسم نمط-على سبيل المثال، COLOR، MENU، (هذا العرف يُطبق على typedefs وstructs، وليس على أسماء الصفوف.) لكن هذا يخلق إمكانية التشابك مع ثوابت "المعالج التمهيدي" المسماة. لتجنب التشابك، بإمكانك ان تجعل بادئة t_ لأسماء الأنماط، مثل t_Color، t_Menu.

ميز الثوابت المسماة تحتاج الثوابت المسماة أن تعرف بحيث يمكنك أن تخبر إن كنت تسند إلى متحول قيمة من متحول آخر (والذي قيمته يمكن أن تتغير) أو من ثابت مسمى. في فيجوال بيسك، لديك الاحتمال الإضافي بأن تكون القيمة من تابع. فيجوال بيسك لا تتطلب أن تستخدم أسماء التوابع الأقواس، بينما في سي++ حتى التوابع التي ليس لها وسطاء تستخدم الأقواس.

أحد النهج في تسمية الثوابت هو أن تستخدم بادئة مثل c_ لأسماء الثوابت. هذا سيعطيك أسماء مثل c_RecsMax، c_LinesPerPageMax. في سي++ وجافا، العرف أن تستخدم كل الأحرف كبيرة، واحتمال استخدام "الشحطة السفلية" لتفصل بين الكلمات، RECSMAX أو RECS_MAX وLINESPERPAGEMAX أو LINES_PER_PAGE_MAX.

ميز عناصر الأنماط التعدادية تحتاج عناصر الأنماط التعدادية أن تعرف لنفس أسباب حاجة الثوابت المسماة- ليكون من الأسهل أن تخبرنا أن الاسم لنمط تعدادي عكس أن يكون لمتحول أو ثابت مسمى أو تابع. ينص النهج المعياري: بإمكانك استخدام كل الأحرف كبيرة أو بادئة e_ او E_ لاسم النمط نفسه وبادئة معتمدة على النمط المحدد مثل Color_، Planet_ لعناصر النمط.

ميز وسطاء "الدخل فقط" في اللغات التي لا تجبر على استخدامها أحيانا يتم تعديل وسطاء الدخل دون قصد في لغات مثل سي++ وفيجوال بيسك، يجب عليك بالتأكيد أن تحدد بشكل صريح إن كنت تريد للقيمة التي تم تعديلها أن ترد إلى الإجرائية المستدعية. هذا يحدد بالمعدلات * و& وconst في سي++ او ByRef وByVal في فيجوال بيسك.

في لغات أخرى[1]، إذا عدلت متحول الدخل، سيتم إرجاعه إن كنت تحب ذلك أو لا. هذا صحيح بشكل خاص عند تمرير الكائنات. في جافا، مثالاً، كل الكائنات تُمرر "بالقيمة"، لذا عندما تمرر كائن إلى إجرائية، فإن محتوى الكائن يمكن أن يتغير ضمن الإجرائية المستدعاة (أرنولد، غوسلينغ، هولمز 2000).

في هذه اللغات[2]، إذا تبنيت عرف تسمية فيه تُعطى وسطاء "الدخل فقط" بادئة const (وfinal أو nonmodifiable أو شيء مشابه)، ستعلم أن خطأ ما قد حدث إذا رأيت أي شيء ببادئة const في الجهة اليسارية من إشارة المساواة. إذا رأيت constMax.SetNewMax(...) ستعلم أن هذا بلاهة لإن بادئة const تدل أن المتحول لا يفترض أن يُعدل.

نسق الأسماء لتحسن إمكانية القراءة تقنيتان شائعتان لزيادة إمكانية القراءة هما استخدام الأحرف الكبيرة والمحارف المباعدة لفصل الكلمات. مثالاً، GYMNASTICSPOINTTOTAL أصعب قراءة من

[1] **توضيح** *بالنسبة لتمرير الكائنات "بالقيمة" في جافا هناك نقطة غير موضحة في هذه الفقرة،* عندما تقوم بإسناد شيء ما (كائن آخر أو نسخة جديدة) إلى متحول الكائن نفسه، داخل الإجرائية المستدعاة، فلن يتغيير الكائن الأصلي (وهذا معنى "بالقيمة"). لكن قيمة الكائن تدل على أعضاء الكائن نفسه، وبالتالي إي تعديل على أعضاء الكائن في الإجرائية المستدعاة سيؤثر على الكائن الأصل. هذا ينطبق على سي# أيضاً. تم اختبار شفرة للتأكد من هذه النتيجة بلغة سي#.

[2] **إشارة مرجعية** تزويد لغة بعرف تسمية للتعويض عن المحدودية في اللغة نفسها هو مثال عن البرمجة في لغة بدلاً من البرمجة بلغة فقط. لتفاصيل أكثر عن البرمجة في لغة، راجع القسم 34. 4، "برمج في لغتك، ليس بها"

gymnasticsPointTotal أو gymnastics_point_total. سي++ وجافا وفيجوال بيسك ولغات أخرى تسمح بخلط المحارف الكبير والصغيرة. وتسمح أيضاً باستخدام الفاصل (_) "الشحطة السفلية"

حاول ألا تخلط بين هذه التقنيات؛ ذلك يجعل الشفرة أصعب قراءةً. إذا قمت بمحاولة نزيهة لاستخدام أي من تقنيات إمكانية القراءة هذه بتماسك، على أية حال، ستحسن شفرتك. تمت سياسة الناس ليكون لديهم جدالات متعصبة وغاضبة حول النقاط الحسنة مثل إن كان المحرف الأول في الاسم ينبغي أن يكون كبيراً (TotalPoints vs totalPoints)، لكن طالما أنك وفريقك متماسكون، لن يصنع استخدام أي منهما فرقاً كبيراً. هذا الكتاب يستخدم البدء بأحرف صغيرة بسبب قوة انتشار جافا ولييسر التشابه في الأسلوب عبر اللغات الأخرى.

توجيهات للأعراف "الخاصة باللغة"

اتبع أعراف تسمية اللغة التي تستخدمها، بإمكانك إيجاد كتب لمعظم اللغات تصف توجيهات أسلوب الكتابة. التوجيهات الخاصة بـ سي وسي++ وجافا وفيجوال بيسك موجودة في الأقسام التالية:

أعراف سي

عدة أعراف تسمية تُطبق بشكل خاص على لغة البرمجة سي[1]:

- c وch هي متحولات محرفية
- i وj هي متحولات صحيحة
- n هو عدد شيء ما
- p هو مؤشر
- s هو سلسلة نصية
- مسجلات "ماكروات" المعالج التمهيدي هي بحالة ALL_CAPS. وهذا عادة يُمدد ليتضمن "تعاريف الأنماط" على حد سواء.
- أسماء الإجرائيات والمتحولات بحالة all_lowercase.
- محرف "الشحطة السفلية" (_) يستخدم كفاصل: letters_in_lowercase هي أسهل قراءةً من lettersinlowercase.

هذه هي الأعراف العامة للبرمجة في سي بأسلوب يونكس وأسلوب لينوكس، لكن أعراف سي مختلفة في بيئات أخرى. في مايكروسوفت ويندوز، مبرمجو سي يميلون لاستخدام تشكيلة من عرف التسمية الهنغاري وحالة مختلطة من الأحرف الصغيرة والكبيرة لأسماء المتحولات. في ماكنتوش، مبرمجو سي يميلون إلى استخدام

[1] اقرأ أيضاً الكتاب التقليدي لأسلوب البرمجة ب سي هو "C Programming Guidelines (Plum 1984)"

"حالة أحرف مختلطة" لأسماء الإجرائيات لأن شريط أدوات ماكنتوش وإجرائيات نظام التشغيل صُممت بالأصل من أجل واجهة باسكال.

أعراف سي++

هنا الأعراف التي نمت حول برمجة سي++[1]:

- i وj هي فهارس صحيحة
- p هو مؤشر
- الثوابت وتعاريف الأنماط وماكروات المعالج التمهيدي بحالة ALL_CAPS
- أسـماء الصـفوف والأنماط الأخرى هي بحالة MixedUpperAndLowerCase (في الكتاب موجودة MixedUpperAndLowerCase() ومما لا شك فيه أن الأقواس زائدة فهي لا تستخدم لأسماء الصفوف أبداً)
- أسماء المتحولات والتوابع تستخدم الأحرف الصغيرة للكلمة الأولى، مع أول حرف من كل كلمة لاحقة كبير-مثالاً، variableOrRoutineName
- لا تستخدم الشحطة السفلية كفا صل ضمن الأ سماء، با ستثناء الأ سماء التي بحالة كل الأحرف كبيرة وأنواع محددة من البوادئ (مثل تلك المستخدمة لتعريف المتحولات الشاملة)

كما هو الحال في برمجة سي، هذا العرف بعيد عن المعيار والبيئات المختلفة صممت معايير بتفاصيل عرفية مختلفة.

أعراف جافا

على نقيض سي وسي++، أعراف أسلوب جافا تأسست بشكل جيد من بدايات اللغة[2]:

- i وj هي فهارس صحيحة
- الثوابت تكون بحالة ALL_CAPS مفصولة بشحطة سفلية
- أسـماء الصـفوف والواجهات بحالة الحرف الأول من كل كلمة كبير، والكلمة الأولى ضـمناً-مثالاً، ClassOrInterfaceName
- أسماء المتحولات والإجرائيات تستخدم الأحرف الصغيرة للكلمة الأولى، مع الحرف الأول من كل كلمة لاحقة كبير-مثالاً، variableOrRoutineName

---

[1] **اقرأ أيضاً** للمزيد من أسلوب البرمجة ب سي++، انظر "عناصر أسلوب سي++ (ميسفيلدت وبومغاردنير وغراي 2004)"

[2] **اقرأ أيضاً** لمزيد حول أسلوب برمجة جافا، انظر "عناصر أسلوب جافا، النسخة الثانية. (فيرميولين إت أل 2000).

- الشحطة السفلية لا تستخدم كفاصل ضمن الأسماء باستثناء الأسماء بحالة كل الأحرف كبيرة
- البادئتان set وget تُستخدمان في طرق الدخول

أعراف فيجوال بيسك

لم تأسس فيجوال بيسك فعلياً أعراف ثابتة. القسم التالي يقدم توصية بعرف لفيجوال بيسك.

أعراف برمجة اللغات المختلطة

عند البرمجة في بيئة مختلطة اللغات، أعراف التسمية (بالإضافة إلى أعراف التنسيق والتوثيق والأعراف الأخرى) يمكن أن تُحسن من أجل سهولة القراءة والتماسك الكلي-حتى إن كانت تخالف عرف واحدة من اللغات التي هي جزء من الخليط.

في هذا الكتاب، على سبيل المثال، أسماء المتحولات كلها تبدأ بأحرف صغيرة، وهذا متناغم مع التطبيق العرفي لبرمجة جافا وبعض لكن ليس كل أعراف سي++. ينسق هذا الكتاب كل أسماء الإجرائيات بأحرف ابتدائية كبيرة، والذي يتبع عرف سي++. عرف جافا أن تبدأ أسماء الطرق بأحرف صغيرة، لكن هذا الكتاب يستخدم أسماء إجرائيات تبدأ بأحرف كبيرة لكل لغات البرمجة لغاية سهولة القراءة الكلية.

أعراف تسمية النماذج

الأعراف المعيارية المذكورة أعلاه تميل لتتجاهل عدة مفاهيم هامة للتسمية تمت مناقشتها قبل صفحات قليلة-متضمنة مجال المتحول (خاص أو صف أو شامل)، هذه المفاهيم تباين بين أسماء الصفوف والكائنات والإجرائيات والمتحولات، والأمور الأخرى.

توجيهات أعراف التسمية يمكن أن تبدو معقدة عندما تصطف عبر عدة صفحات. إنها لا تحتاج أن تكون معقدة بشكل مزعج، على أية حال، بإمكانك أن تكيفها وفق حاجاتك. أسماء المتحولات تحوي ثلاث أنواع من المعلومات:

- محتوى المتحول (ماذا يمثل)
- نوع البيانات (ثابت مسمى أو متحولات بدائية أو نمط معرف من قبل المستخدم أو صف)
- مجال المتحول (خاص أو صف أو حزمة أو شامل)

الجداول 11-3 و11-4 و11-5 تقدم أعراف تسمية لـ سي وسي++ وجافا وفيجوال بيسك والتي تم تكييفها لتناسب التوجيهات المقدمة سابقاً. هذه الأعراف الخاصة ليست توصيات ضرورية، لكنها تقدم لك فكرة عم يحوي عرف التسمية غير الرسمي.

جدول 11-3 أعراف تسمية النماذج لجافا وسي++

الكيان	الوصف

ClassName	أسماء الصفوف بحالة حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي كبير
TypeName	تعريفات الأنماط، متضمنةً الأنماط التعدادية وتعاريف الأنماط(typedefs)، تستخدم حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي كبير
EnumeratedTypes	بالإضافة إلى القاعدة السابقة، الأنماط التعدادية دائماً تكون بصيغة الجمع.
localVariable	المتحولات المحلية بحالة حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي صغير. ينبغي أن يكون الاسم مستقل عن نوع البيانات القائم عليه وينبغي أن يشير إلى كل شيء يمثله المتحول
routineParameter	وسطاء الإجرائية تُنسق بنفس طريقة المتحولات المحلية
()RoutineName	الإجرائيات بحالة حروف كبيرة وصغيرة مختلطة. (أسماء الإجرائيات الجيدة نوقش في القسم 7. 3)
m_ClassVariable	المتحولات الأعضاء التي تكون متاحة لعدة إجرائيات في الصف، لكن فقط ضمن الصف، تبدأ ب *m*_
g_GlobalVariable	المتحولات الشاملة تبدأ ب *g*_
CONSTANT	الثوابت المسماة بحالة *ALL_CAPS*
MACRO	الماكروات بحالة *ALL_CAPS*
Base_EnumeratedType	تبدأ الأنماط التعدادية بكلمة سهلة التذكر تشير إلى النمط الأساسي بصيغة المفرد-مثالاً، *Color_Red*، *Color_Blue*

جدول 11-4 أعراف تسمية النماذج للغة سي

الكيان	الوصف
TypeName	تعاريف الأنماط تستخدم حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي كبير
GlobalRoutineName()	الإجرائيات العامة بحالة حروف كبيرة وصغيرة مختلطة
f_FileRoutineName()	الإجرائيات الخاصة بوحدة (ملف) مفردة تبدأ ب f_
LocalVariable	المتحولات المحلية بحالة حروف كبيرة وصغيرة مختلطة. ينبغي أن يكون الاسم مستقل عن نوع البيانات القائم عليه وينبغي أن يشير إلى كل شيء يمثله المتحول
RoutineParameter	وسطاء الإجرائية تُنسق بنفس طريقة المتحولات المحلية
f_FileStaticVariable	متحولات الوحدة (الملف) تبدأ ب f_
G_GLOBAL_GlobalVariable	المتحولات الشاملة تبدأ ب G_ وكلمة سهلة التذكر عن الوحدة (الملف) التي تعرف المتحول بحالة حروف كلها كبيرة-مثالاً، SCREEN_Dimensions
LOCAL_CONSTANT	الثوابت المسماة الخاصة ب إجرائية مفردة أو وحدة مفردة بحالة كل الحروف كبيرة-مثالاً، ROWS_MAX
G_GLOBALCONSTANT	الثوابت المسماة بحالة كل الحروف كبيرة وتبدأ ب G_ وكلمة سهلة التذكر عن الوحدة (الملف) التي تعرفها بحالة كل الحروف كبيرة-مثالاً، G_SCREEN_ROWS_MAX
LOCALMACRO()	تعاريف الماكروات الخاصة بإجرائية مفردة أو وحدة (ملف) مفردة بحالة كل الحروف كبيرة
G_GLOBAL_MACRO()	تعاريف الماكروات الشاملة بحالة كل الأحرف كبيرة وتبدأ ب G_ وكلمة سهلة التذكر عن الوحدة (الملف) التي تعرف الماكرو بحالة كل الأحرف كبيرة-مثالاً، G_SCREEN_LOCATION()

لأن فيجوال بيسك ليست حساسة لحالة الأحرف، فإن قواعد خاصة تُطبق لتمييز أسماء الأنماط وأسماء المتحولات. ألق نظرة على الجدول 11-5.

جدول 11-5 أعراف تسمية النماذج لفيجوال بيسك

الكيان	الوصف
C_ClassName	أسماء الصفوف بحالة حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي كبير وبادئة _C
T_TypeName	تعريفات الأنماط، متضمنةً الأنماط التعدادية وتعاريف الأنماط(typedefs)، تستخدم حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي كبير وبادئة _T
T_EnumeratedTypes	بالإضافة إلى القاعدة السابقة، الأنماط التعدادية دائماً تكون بصيغة الجمع.
localVariable	المتحولات المحلية بحالة حروف كبيرة وصغيرة مختلطة مع حرف ابتدائي صغير. ينبغي أن يكون الاسم مستقل عن نوع البيانات القائم عليه وينبغي أن يشير إلى كل شيء يمثله المتحول
routineParameter	وسطاء الإجرائية تُنسق بنفس طريقة المتحولات المحلية
RoutineName()	الإجرائيات بحالة حروف كبيرة وصغيرة مختلطة. (أسماء الإجرائيات الجيدة نوقش في القسم 7. 3)
m_ClassVariable	المتحولات الأعضاء التي تكون متاحة لعدة إجرائيات في الصف، لكن فقط ضمن الصف، تبدأ ب _m
g_GlobalVariable	المتحولات الشاملة تبدأ ب _g
CONSTANT	الثوابت المسماة بحالة ALL_CAPS
Base_EnumeratedType	تبدأ الأنماط التعدادية بكلمة سهلة التذكر تشير إلى النمط الأساسي بصيغة المفرد-مثالاً، Color_Red، Color_Blue

11. 5 بادئات تابعة للمعايير

تؤمن البادئات "التابعة للمعايير" للمعاني الشائعة نهج مقتضب لكن متماسك وسهل القراءة لتسمية البيانات.[1] أفضل تخطيط معروف للبادئات التابعة للمعايير هو عرف التسمية الهنغاري، والذي هو مجموعة من التوجيهات التفصيلية لتسمية المتحولات والتوابع (وليس الهنغاريين!) والذي استُخدم على نطاق واسع في أحد المرات في برمجة مايكروسوفت ويندوز. على الرغم من أن عرف التسمية الهنغاري لم يعد بذلك الانتشار، الفكرة الأساسية لمَعيَرَة اختصارات بشكل مقتضب ودقيق لا تزال قيمة.

البادئات التابعة للمعايير تتألف من جزأين: اختصارات الأنماط "المعرفة من المستخدم" (User-Defined Types 'UDT') والبادئات الدلالية.

[1] **اقرأ أيضاً** لتفاصيل أكثر حول عرف التسمية الهنغاري، راجع "الثورة الهنغارية" (سيمونيه وهيلر 1991).

اختصارات الأنماط المعرفة من المستخدم

تعرف اختصارات UDT أنماط البيانات للكائنات والمتحولات تحت عملية التسمية. ربما تشير اختصارات UDT إلى كيانات مثل النوافذ ومناطق الشاشة والخطوط. لا تشير UDT بشكل عام إلى أي من أنماط البيانات المعرفة مسبقاً المقدمة من قبل لغة البرمجة.

توصَف UDT بالشفرات القصيرة التي تكتبها من أجل برنامج محدد ثم تمعيرها للاستخدام في ذلك البرنامج. تكون الشفرات سهلة التذكر مثل wn ل windows وscr لـscreen regions. يقدم الجدول 6-11 لائحة نماذج عن UDTs التي يمكن أن تستخدمها في برنامج لمعالجة النصوص.

جدول 6-11 نموذج ل UDTs لمعالج نصوص

اختصارات UDT	المعنى
ch	Character محرف (ليس المحرف الخاص بعالم C++، لكن بعالم نمط بيانات يستخدمه برنامج معالج النصوص ليمثل محرف في مستند)
doc	Document مستند
pa	Paragraph مقطع
sc	Screen region منطقة من الشاشة
sel	Selection الاختيار
wn	Window نافذة

عندما تستخدم UDTs، فإنك تعرف أنماط بيانات "لغة البرمجة" التي تستخدم نفس الاختصارات المستخدمة في UDTs. هكذا، إذا كان لديك UDTs كما في الجدول 6-11، سترى تصريحات بيانات كالتالي:

```
CH     chCursorPosition;
SCR    scrUserWorkspace;
DOC    docActive
PA     firstPaActiveDocument;
PA     lastPaActiveDocument;
WN     wnMain;
```

ثانية، هذه الأمثلة مرتبطة بمعالج النصوص. لاستخدام هكذا اختصارات في مشاريعك الخاصة، أنشئ اختصارات UDT ل UDTs المستخدمة بشكل شائع جداً في بيئتك.

البادئات الدلالية

تخطو البادئات الدلالية خطوة أعمق من UDT وتصف كيف يتم استخدام المتحول أو الكائن. بخلاف UDTs، التي تختلف من مشروع إلى آخر، البادئات الدلالية تكون معيارية بطريقة ما عبر المشاريع. يوضح الجدول 7-11 لائحة من البادئات الدلالية المعيارية.

جدول 7-11 البادئات الدلالية

البادئة الدلالية	المعنى
c	عدد Count (كما في عدد السجلات والمحارف وهلم جراً)
first	العنصر الأول الذي يحتاج أن يتم التعامل معه في مصفوفة. first مشابه ل min لكنه مرتبط بالعملية الحالية بدلاً من المصفوفة نفسها
g	متحول شامل
i	فهرس مصفوفة
last	آخر عنصر يحتاج أن تتم معالجته في مصفوفة. last هو المقابل لfirst
lim	الحد الأعلى للعناصر التي تحتاج أن تتم معالجتها في مصفوفة. lim ليس دليل صالح، إنه يمثل حافة عليا مفتوحة (ليست مضمنة) لمصفوفة؛ last يمثل عنصر أخير قانوني. بشكل عام lim = last +1
m	متحول في مستوى الصف
max	العنصر الأخير المطلق في مصفوفة أو أي نوع من اللوائح. max يشير إلى المصفوفة نفسها بدلاً من العمليات على المصفوفة
min	العنصر الأول المطلق في مصفوفة أو نوع لوائح آخر
p	مؤشر

تُنسق البادئات الدلالية بحالة أحرف صغيرة أو خليط من الأحرف الصغيرة والكبيرة وتدمج مع UDTs والبادئات الدلالية الأخرى وفق الحاجة. مثالاً، ينبغي أن يُسمَّى المقطع الأول في مستند pa ليبين أنه مقطع وfirst ليبين أنه المقطع الأول: firstPa. ينبغي أن يسمى فهرس لمجموعة من المقاطع iPa؛ cPa هو العدد، أو عدد المقاطع؛ وfirstPaActiveDocument وlastPaActiveDocument هما المقطعين الأول والأخير في المستند النشط الحالي.

حسنات البادئات التابعة للمعايير



تعطيك البادئات التابعة للمعايير كل الحسنات العامة لأعراف التسمية بالإضافة إلى عدة حسنات أخرى. لأن الكثير من الأسماء معياري، فلديك القليل من الأسماء لتتذكره في صف أو برنامج واحد.

تضيف البادئات التابعة للمعايير الدقة إلى عدة مناطق من التسمية التي تميل لتكون غير دقيقة. التفريق الدقيق بين min، first، last، max يكون مساعد بشكل خاص.

البادئات التابعة للمعايير تجعل الأسماء مضغوطة أكثر. مثالاً، بإمكانك استخدام cpa لعدد المقاطع بدلاً من totalParagraphs. واستخدام ipa لتعرف فهرس مصفوفة من المقاطع بدلاً من indexParagraphs أو paragraphsIndex.

أخيراً، تتيح لك البادئات التابعة للمعايير اختبار الأنماط بدقة عند استخدامك أنماط البيانات المجردة التي لا يستطيع بالضرورة مترجمك أن يختبرها: paReformat = docReformat من المحتمل أن تكون خطأ لأن pa وdoc هما UDT مختلفان.

الفخ الرئيسي في البادئات التابعة للمعايير، أن المبرمج يستخف بإعطاء المتحولات أسماء ذات معنى بوجود البادئات فيها. إن كان ipa يمثل بلا غموض فهرس مصفوفة مقاطع، فإنه لا يحاول أن يجعل الاسم أكثر دلالة مثل ipaActiveDocument. لسهولة القراءة، أغلق الدارة وائتِ بأسماء معبرة.

11. 6 إنشاء أسماء قصيرة تكون مقروءة

ترجع الرغبة باستخدام أسماء متحولات قصيرة بطريقة أو أخرى إلى عصر الحوسبة الأول. اللغات القديمة مثل assembler وBasic generic وFortran حددت طول أسماء المتحولات ب2-8 محارف وأجبرت المبرمجين على إنشاء أسماء قصيرة. الحوسبة الأولى (زماناً) كانت مرتبطة جداً بالرياضيات واستخداماتها لمصطلحات مثل i، j، k كمتغيرات في عمليات الجمع والمعادلات الأخرى. في اللغات الحديثة مثل سي++ وجافا وفيجوال بيسك، يمكنك أن تنشئ فعلياً أسماء بأي طول، لا يوجد في معظم الأحيان سبب لتقصير الأسماء ذات الدلالة.

إذا تطلبت الظروف أن تنشئ أسماء قصيرة، لاحظ أن بعض الطرق لتقصير الأسماء أفضل من بعض. بإمكانك إنشاء أسماء جيدة وقصيرة للمتحولات بالاستغناء عن الكلمات عديمة الفائدة واستخدام مرادفات أقصر واستخدام أي من استراتيجيات الاختصارات المتعددة. إنها لفكرة جيدة أن تكون متآلف مع التقنيات المتعددة للاختصارات لأنه لا توجد تقنية مفردة تعمل بنجاح في كل الحالات.

توجيهات عامة للاختصارات

هنا عدة توجيهات لإنشاء الاختصارات. بعضها يتعارض مع بعض، لذا لا تحاول أن تستخدمها كلها بنفس الوقت.

- استخدم الاختصارات المعيارية (الاختصارات شائعة الاستخدام، التي تكون مرتبة في القاموس)
- أزل كل الأحرف الصوتية غير الابتدائية. (computer تصبح cmptr، screen تصبح scrn، apple تصبح appl، integer تصبح intgr).
- أزل الأدوات اللغوية مثل: and، or، the وما نحى نحوها
- استخدم الحرف الأول أو الحروف القليلة الأولى من كل كلمة
- اقطع بثبات بعد الحرف الأول أو الثاني أو الثالث (أي واحد مناسب) من كل كلمة
- حافظ على الحرفين الأول والأخير من كل كلمة
- استخدم كل كلمة مهمة في الاسم، وصولاً إلى ثلاث كلمات كحد أقصى
- أزل اللاحقات عديمة الفائدة-ing، ed وما نحى نحوها
- حافظ على الصوت الأكثر تميزاً في كل مقطع صوتي في الكلمة

- تأكد من ألا تغير معنى المتحول
- كرر هذه التقنيات حتى تختصر كل اسم متحول ليكون بطول بين 8 و20 محرف أو إلى عدد المحارف المحدد من قبل لغتك.

الاختصارات الصوتية

يدافع بعض الناس عن إنشاء اختصارات قائمة على أساس لفظ (صوت) الكلمات بدلاً من هجائها (حروفها). وهكذا skating تصبح sk8ing وhighlight تصبح hilite وbefore تصبح b4 وexcute تصبح xqt وهكذا. هذا يشبه كثيراً سؤال الناس أن يكتشفوا "الألواح المعدنية"[1] الخاصة بي. ولا أنصح بها. كتمرين اكتشف ماذا تعني هذه الأسماء:

ILV2SK8 XMEQWK S2DTM8O NXTC TRMN8R

التعليقات على الاختصارات

يمكن أن تسقط في عدة حفر أثناء إنشاء الاختصارات. هنا بعض القواعد لتجنب الفخاخ:

لا تختصر بإزالة محرف واحد من الكلمة كتابة حرف واحد هو عمل إضافي خفيف، وادخارات الحرف الواحد تُبرِز بقسوة خسارة سهولة القراءة. إنها مثل المفكرات التي تحتوي على Jun وJul عليك أن تكون بعجلة فائقة حتى تلفظ June مثل Jun. مع معظم الإزالات للحرف الواحد، من الصعب أن تتذكر إن أزلت حرفاً. أو أزلت أكثر من حرف واحد أو هجّأت الكلمة (الإشارة هنا إلى الاختصارات الصوتية).

اختصر بتماسك استخدم دائماً نفس الاختصارات. مثالاً، استخدم Num في كل مكان أو No في كل مكان، لكن لا تستخدمهما معاً. بشكل مشابه، لا تختصر كلمة في بعض الأسماء وتبقيها على حالها في أسماء أخرى. مثالاً، لا تستخدم كامل الكلمة Number في أماكن ما وتختصرها إلى Num في أماكن أخرى.

أنشئ أسماء تستطيع لفظها استخدم xPos بدلاً من xPstn، وneedsComp بدلاً من ndsCmptg. طبق اختبار الهاتف-إذا لم تستطع قراءة شفرتك لشخص ما عبر الهاتف، أعد تسمية متحولاتك لتكون أكثر تمييزاً (كيرنيغان وبلاوغير 1978).

تجنب الدمج الذي يسبب اختلاط بالقراءة أو باللفظ للإشارة إلى نهاية B، فضل ENDB على BEND. إذا كنت تستخدم تقنية فصل جيدة، لن تحتاج هذا التوجيه منذ أن B-END وBEnd وb_end لن يختلط لفظها.

[1] **مصطلحات** اللوحة المعدنية (license plate) هي لوحة معدنية ينقش عليها رقم التسجيل الخاص بالسيارة

استخدم قاموساً لتحل تضارب الأسماء إحدى المشاكل في إنشاء أسماء قصيرة هو تصادم التسميات-تُختصر الأسماء إلى نفس الشيء. مثالاً، إذا كنت محدوداً بثلاثة محارف وتحتاج أن تستخدم fired و full revenue disbursal في نفس المنطقة من البرنامج، ربما بشكل غير ملائم ستختصر الإثنين إلى frd.

لتجنب هذا بطريقة سهلة، استخدم كلمات مختلفة لنفس المعنى، لذا القاموس مفيد! في هذا المثال، dismissed يمكن أن تكون بديل ل fired، وcomplete revenue disbursal يمكن أن تكون بديل لfull revenue disbursal. والاختصارين ثلاثيي الأحرف يصبحان dsm وcrd، ويزول تصادم التسمية.

وثق بشدة الأسماء القصيرة بجداول ترجمة في الشفرة في اللغات التي تسمح بأسماء قصيرة جداً، ضمّن جدول ترجمة لتؤمن تذكار بالكلمات المساعدة على التذكر في المتحولات. ضمّن الجدول كتعليق في بداية كتلة من الشفرة. هنا مثال:

مثال بلغة فورتران عن جدول ترجمة جيد

```
C *******************************************************************
C    Translation Table
C
C    Variable Meaning
C    -------- -------
C    XPOS x-Coordinate Position (in meters)
C    YPOS Y-Coordinate Position (in meters)
C    NDSCMP     Needs Computing (=0 if no computation is needed;
C    =1 if computation is needed)
C    PTGTTL Point Grand Total
C    PTVLMX Point Value Maximum
C    PSCRMX     Possible Score Maximum
C *******************************************************************
```

ربما تفكر أن هذه التقنية منتهية الصلاحية، لكن بحداثة منتصف 2003 قمت بالعمل مع زبون لديه المئات من الآلاف من الأسطر المكتوبة بRPG. والتي كانت خاضعة لأسماء متحولات بحد أعلى ستة حروف. هذه القضايا تظهر من حين لآخر.

وثق كل الاختصارات بوثيقة "اختصارات معيارية" على مستوى المشروع الاختصارات في الشفرة تخلق خطرين عامّين:

- قد لا يفهم قارئ الشفرة الاختصار

- قد يســـتخدم مبرمجون آخرون عدة اختصـــارات ليشـــيروا إلى نفس الكلمة، والذي يخلق ارتباك غير ضروري

لمعالجة كلا المشكلتين المحتملتين، بإمكانك إنشاء وثيقة "الاختصارات المعيارية" التي تلتقط كل اختصارات كتابة الشفرة المستخدمة في مشروعك. يمكن أن تكون وثيقة معالج نصوص أو جداول إلكترونية. في المشاريع الكبيرة جداً يمكن أن تكون وثيقة قاعدة بيانات. الوثيقة تخضع ل "التحكم بالإصدار"[1] وتراجع في أي وقت من قبل أي شخص ينشئ اختصارات جديدة في الشفرة. المداخل في هذه الوثيقة ينبغي أن ترتب حسب الكلمة الكاملة، وليس الاختصار.

قد يبدو هذا كالكثير من الحمل الزائد، لكن بجانب مقدار صغير من الحمل الزائد في التأسيس، فإنه فعلياً يؤسس آلية تساعد المشروع على استخدام الاختصارات بفعالية. إنه يعالج الخطر الأول من الخطرين العامين المذكورين في الأعلى بتوثيق كل الاختصارات المستخدمة. حقيقة أن المبرمج لا يستطيع إنشاء اختصار جديد بدون العمل الزائد بفحص وثيقة الاختصارات المعيارية الصادرة عن "المتحكم بالإصدارات"، وإدخال الاختصار، واختبارها مجدداً هي شيء حسن. هذا يعني أن الاختصار لن ينشأ مالم يكن منتشر جداً ليستحق الصراع لتوثيقه.

يعالج هذا النهج الخطر الثاني بتقليل احتمالية إنشاء المبرمجين لاختصارات متكررة. المبرمج الذي يريد أن يختصر شيء ما سيفحص وثيقة الاختصارات ويدخل الاختصار الجديد. إذا كان هناك مسبقاً اختصار للكلمة التي يريد أن يختصرها، سيلاحظ المبرمج ذلك ويستخدم الاختصار الموجود بدلاً من إنشاء واحد جديد.



القضية العامة المشروحة في هذا التوجيه هي الفرق بين "أريحية وقت-الكتابة" و"أريحية وقت-القراءة". هذا النهج يخلق بوضوح "إزعاج وقت- الكتابة"، لكن المبرمجين عبر زمن حياة نظام ما يصرفون وقت أكبر بكثير وهم يقرؤون الشفرة منه وهم يكتبون شفرة. هذا النهج يزيد "أريحية وقت-القراءة". مع مرور الوقت كل الغبار في المشروع سيسكن، هذا قد يحسن أيضاً "أريحية وقت-الكتابة."

تذكر أن الأسماء تهم قارئ الشفرة أكثر من الكاتب اقرأ شفرة من إنتاجك لم ترها منذ ستة أشهر على الأقل ولاحظ أين يتوجب عليك العمل كي تفهم ما تعني الأسماء. اعزم على تغيير الممارسات التي سببت هكذا ارتباكاً.

11.7 أنواع من الأسماء للتجنب

هنا بعض التوجيهات المتعلقة بأسماء متحولات عليك أن تتجنبها:

[1] **مصطلحات** "التحكم بالإصدار" تقنية لضبط التغيرات في كتاب أو برنامج أو ما أشبه، بحيث يعطى كل تغيير جديد رقم اصدار خاص. ويرفق مع كل اصدار طابع زمني، واسم الشخص الذي قام بالإصدار.

تجنب الأسماء أو الاختصارات التي يساء فهمها تأكد من أن الاسم غير غامض. على سبيل المثال، FALSE هي عادة نقيض TRUE وستكون اختصار سيء لـ"Fig and Almond Season."

تجنب الأسماء ذات المعاني المتشابهة إذا كان بإمكانك أن تبدل اسمي متحولين دون أن تؤذي البرنامج، فعليك أن تعيد تسمية المتحولين. مثالاً، input وinputValue، recordNum وnumRecords، fileNumber وfileIndex، كلها متشابه جداً من الناحية الدلالية بحيث إذا استخدمتها في نفس الموضع من الشفرة ستخطأ بينهما بسهولة وستقوم بتركيب أخطاء صعبة الإيجاد والملاحظة.

تجنب المتحولات ذات المعاني المختلفة والأسماء المتشابهة[1] إذا كان لديك متحولين باسمين متشابهين ومعنيين مختلفين، حاول أن تعيد تسمية واحد منهما أو أن تغير اختصاراتك. تجنب أسماء مثل clientRecs وclientReps. إنهما مختلفان بحرف واحد فقط عن بعضهما البعض، والحرف صعب الملاحظة. ليكن لديك اختلافات بحرفين على الأقل بين الأسماء، أو ضع الاختلافات في البداية أو النهاية. clientRecords وclientReports أفضل من الاسمين الأولين.

تجنب الأسماء التي تُسمع بشكل متشابه، مثل wrap وrap نجد الكلمات المتماثلة باللفظ في الطريق أمامنا عندما نحاول أن نناقش الشفرة مع الآخرين. واحدة من مثيرات غضبي حول "البرمجة القصوى" (بك 2000) هي استخدامها الفائق الذكاء لمصطلحي Goal Donor وGold Owner، والتي هي فعلياً غير قابلة للتمييز عندما تُنطق. سنصل إلى محادثة كهذه:

I was just speaking with the Goal Donor—
Did you say "Gold Owner" or "Goal Donor"?
I said "Goal Donor."
What?
GOAL - - - DONOR!
OK ،Goal Donor. You don't have to yell ،Goll' Darn it.
Did you say "Gold Donut?"

تذكر أن اختبار الهاتف يُطبق إلى الأسماء المتشابهة باللفظ تماماً مثل الأسماء المختصرة بشكل شاذ.

تجنب الأرقام في الأسماء إذا كانت الأرقام في الاسم مهمة فعلاً، استخدم مصفوفة بدلاً من متحولات متفرقة. إن كانت المصفوفة غير مناسبة، الأرقام هي أبعد عن المناسبة. مثلاً، تجنب file1، file2 أو total1، total2. تستطيع تقريباً دائماً التفكير بطريقة أفضل للتمييز بين متحولين من دقّ 1 أو 2 في نهاية الاسم. لا

[1] **إشارة مرجعية** المصطلح التقني لفوارق مثل هذه بين أسماء المتحولات المتشابهة هو "المسافة النفسية" لتفاصيل، راجع "كيف يمكن للمسافة *النفسية* أن تساعد" في القسم 23. 4

أستطيع القول لا تستخدم الأرقام مطلقاً. بعض كيانات العالم الحقيقي (مثل Route 66 أو Interstate 405) لديها أرقام مضمنة بها. لكن ضع بحسبانك إن كان هناك بدائل أفضل قبل أن تنشئ اسماً يحوي أرقاماً.

تجنب الكلمات التي تُهجأ خطأً إنه صعبٌ كفاية أن تتذكر كيف يُفترض أن تُهجأ الكلمات. أن تتطلب من الناس أن يتذكروا التهجئة الخاطئة "الصحيحة" هو ببساطة شيء كثير على سؤال. مثلاً، تهجئة highlight خطأً إلى hilite لنوفر ثلاثة محارف، تصعب على القارئ بشكل شيطاني أن يتذكر كيف تمت تهجئتها خطأً، أكانت highlite? hilite? hilight? hilit? jai-a-lai-t? من يعلم؟

تجنب الكلمات التي تهجأ خطأً بشكل شائع في اللغة الإنكليزية Absense، acummulate، acsend، calender، concieve، defferred، definate، independance، occassionally، prefered، reciept، superseed والكثير أيضاً هي تهجئات خاطئة شائعة في الإنكليزية. الكثير من كتب اليد الخاصة باللغة الإنكليزية تحتوي لائحة من الكلمات التي تهجأ خطأً بشكل شائع. تجنب استخدام هكذا كلمات في أسماء متحولاتك.

لا تباين أسماء المتحولات بتكبير الأحرف فقط إذا كنت تبرمج بلغة حساسة لحالة الأحرف مثل سي++، قد تُفتن باستخدام frd ل fired، وFRD ل final review duty، وFrd ل full revenue disbursal. تجنب هذا التطبيق. مع أن الأسماء متباينة، فإن ربط كل واحد منه بمعنى محدد هو أمر اعتباطي ومربك. Frd يمكن بسهولة تامة أن تكون مرتبطة ب final review duty، وFRD ب full revenue disbursal، ولا توجد قاعدة تساعدك أنت أو أي شخص آخر لتذكر من هو من.

تجنب تعدد اللغات الطبيعية في المشاريع متعددة الجنسيات، أجبر استخدام لغة طبيعية واحدة لكل الشفرة، متضمناً أسماء الصفوف وأسماء المتحولات وهلم جراً. قراءة شفرة مبرمج آخر يمكن أن تكون تحدٍ؛ قراءة شفرة مبرمج آخر بالمريخية الجنوبية أمر مستحيل.

تحدث مشكلة أكثر مكراً في نسخ اللغة الإنكليزية، إذا كان المشروع مداراً في عدة بلدان ناطقة بالإنكليزية، اجعل واحدة من إصدارات الإنكليزية معيارية، عندها لن تتساءل دائماً إن كان ينبغي ان تقول الشفرة "color" أو ""colour، check"" أو "cheque" وهلم جراً.

تجنب أسماء الأنماط والمتحولات والإجرائيات القياسية كل دليل للغات البرمجة يحتوي لوائح من الأسماء المحجوزة والمعرفة من قبل اللغة. اقرأ دليل اللغة من حين لآخر لتتأكد أنك لا تطأ على قدمي اللغة التي



تستخدمها. مثلاً، مقطع الشفرة التالي هو مسموح بلغة PL/I، لكن فقط إن كنت مغفلاً جاهزاً للترخيص استخدمه:

```
if if = then then
then = else;
else else = if;
```

لا تستخدم أسماء لا تمت بصلة إلى ما يمثل المتحول الأسماء كـ margaret وpookie المبعثرة في برنامج تضمن فعلياً ألا أحد غيرك سيكون قادراً على فهم ما تعني. تجنب اسم صديقك أو زوجتك أو جعتك المفضلة أو أي أسماء ذكية (وتعرف أيضا ب الغبية) للمتحولات، ما لم يكن البرنامج فعلياً عن صديقك أو زوجتك أو جعتك المفضلة. وعندها حتى، ستتعرف ان كنت حكيماً أن كل منها قد يتغير، ولذلك الأسماء العامة boyfriend، wife، favoriteBeer هي أفضل.

تجنب الأسماء المتضمنة محارف صعبة القراءة كن حذراً من أن بعض المحارف تبدو متشابهة جداً والتي من الصعب أن نفرقها عن بعضها. إذا كان الفرق الوحيد بين اسمين هو واحد من هذه المحارف، ربما سيكون لديك وقت صعب لتميز الأسماء عن بعضها. مثلاً، حاول أن تؤشر على الاسم الغريب في كل من المجموعات التالية

```
eyeChartl eyeChartI eyeChartl
TTLCONFUSION     TTLCONFUSION     TTLC0NFUSION
hard2Read hardZRead hard2Read
GRANDTOTAL GRANDTOTAL 6RANDTOTAL
ttl5  ttlS  ttlS
```

تشتمل الأزواج الصعبة التمييز ((1 and l)، (1 and I)، (. and ,)، (0 and O)، (2 and Z)، (; and:)، (S and 5)، (G and 6)

هل تفاصيل مثل هذه تشكل مشكلة حقيقةً؟[1] بالتأكيد! جيرالد ويببيرغ أصدر تقريراً أنه بين 1970 و1980، فاصلة استخدمت في عبارة FORMAT في الفورتران حيث كان ينبغي أن تستخدم نقطة. كانت النتيجة أن العلماء حسبوا خطأً مسار مركبة فضائية وتمت خسارة المسبار الفضائي-بما يطرب الأذن من 1.6 بليون دولار (وينبيرغ 1983).

لائحة اختبار: تسمية المتحولات2

[1] **إشارة مرجعية** لاعتبارات استخدام البيانات، انظر قائمة التحقق في الصفحة 384 في الفصل 10، "قضايا عامة في استخدام المتغيرات".

[2] cc2e.com/1191

اعتبارات تسمية عامة

- هل يشير الاسم إلى المشكلة الواقعية بدلاً من الحل بلغة البرمجة؟
- هل الاسم طويل كفاية حيث لا تحتاج أن تحزر ما هو؟
- هل معدلات "القيم المحسوبة"، إن وجدت، في نهاية الاسم؟
- هل يستخدم الاسم Count أو Index بدلاً من Num؟

تسمية أنواع محددة من البيانات

- هل أسماء فهارس الحلقات ذات معنى (شيء ما غير i,j,k إذا كانت الحلقة أكثر من سطر أو سطرين طولاً أو كانت معششة؟
- هل تمت إعادة تسمية المتحولات "المؤقتة" إلى شيء ما أكثر تعبيراً؟
- هل تمت تسمية المتحولات المنطقية بحيث تكون معانيها عندما تكون true واضحة؟
- هل تحتوي أسماء الأنماط التعدادية على بادئة أو لاحقة تحدد الفئة-مثالاً، Color_ من أجل Color_Red وColor_Green وColor_Blue وما إلى هنالك؟
- هل تمت تسمية الثوابت المسماة لتدل على الكيانات المجردة التي تمثلها بدلاً من الأرقام التي تشير إليها؟

أعراف التسمية

- هل يميز العرف بين البيانات الشاملة والخاصة بالصف والمحلية؟
- هل يميز العرف بين أسماء الأنماط والثوابت المسماة والأنماط التعدادية والمتحولات؟
- هل يعرّف العرف وسطاء" الدخل فقط" للإجرائية في اللغات التي لا تفرض استخدامها؟
- هل العرف متوافق ما أمكن مع الأعراف المعيارية للّغة؟
- هل تم تنسيق الأسماء لتسهيل القراءة؟

الأسماء القصيرة

- هل تستخدم الشفرة أسماء طويلة (إلا إذا كان استخدام أسماء قصيرة ضرورياً)؟
- هل تتجنب الشفرة الاختصارات التي تحتصر فقط حرف واحد من كامل الكلمة؟
- هل تم اختصار كل الكلمات بشكل متماسك؟
- هل الأسماء قابلة للّفظ؟
- هل تم تجنب الأسماء التي يمكن أن يساء فهمها أو قراءتها؟
- هل تم توثيق الأسماء القصيرة في جداول ترجمة؟

مشاكل تسمية شائعة: هل تجنبت...

- ...الأسماء التي تُفهم، غير ما تعني؟
- ...الأسماء ذات المعاني المتشابهة؟
- ...الأسماء التي تتباين بمحرف واحد أو محرفين؟
- ...الأسماء التي تتشابه باللفظ؟
- ...الأسماء التي تستخدم الأرقام؟
- ...الأسماء التي تُهجئ بطريقة خاطئة عمداً كي تصبح أقصر؟
- ...الأسماء التي بشكل شائع تُهجئ بطريقة خاطئة في اللغة الإنكليزية؟
- ...الأسـماء التي تتضـارب مع أسـماء مكتبة الإجرائيات المعيارية أو مع أسـماء المتحولات المعرفة مسبقاً؟
- ...الأسماء "العشوائية بشكل كامل"؟
- ...المحارف صعبة القراءة؟

نقاط مفتاحية

- أسماء المتحولات الجيدة هي عنصر أساسي لإمكانية قراءة برنامج. أنواع محددة من المتحولات مثل فهارس الحلقات ومتحولات الحالة تحتاج اعتبارات محددة.
- ينبغي أن تكون الأســماء محددة قدر الإمكان. الأســماء التي تكون غامضــة كفاية أو عامة كفاية لتستخدم لأكثر من غرض واحد هي عادة أسماء سيئة.
- تميز أعراف التســمية بين البيانات الشــاملة والخاصــة بالصــف والمحلية. وتميز بين اســماء الأنماط والثوابت المسماة والأنماط التعدادية والمتحولات.
- بصـرف النظر عن نوع المشـروع الذي تعمل عليه، ينبغي أن تتبنى عرف تسـمية متحولات. نوع العرف الذي تتبناه يعتمد على حجم برنامجك وعدد الناس العاملين فيه.
- نادراً ما نحتاج الاختصــارات في لغات البرمجية الحديثة. إذا كنت حقاً تســتخدم الاختصــارات، تتبع الاختصارات في قاموس المشروع أو استخدم نهج البادئات التابعة للمعايير.
- تُقرأ الشفرة مرات أكثر بكثير مما تُكتب. تأكد أن الأسماء التي تختارها تفضّل "أريحية وقت-القراءة" على "أريحية وقت-الكتابة."

الفصل 12

أنواع البيانات الأساسية

المحتويات[1]

مواضيع ذات صلة

- تسمية البيانات: الفصل 11
- أنواع البيانات غير العادية: الفصل 13
- قضايا عامة في استخدام المتغيرات: الفصل 10
- تنسيق التصريحات عن البيانات: "تخطيط التصريحات عن البيانات": في القسم 31. 5
- توثيق المتغيرات: "التعليق على التصريح عن البيانات" في القسم 32. 5
- الصفوف الناجحة: الفصل 6

إن أنواع البيانات(المعطيات) الأساسية هي كتل البناء الأساسية لجميع أنواع المعطيات الأخرى. يحتوي هذا الفصل على نصائح لاستخدام الأعداد (بشكل عام)، والأعداد الصحيحة، والأعداد الحقيقية، والمحارف

[1] cc2e.com/1278

والسلاسل المحرفية(النصيّة)، والمتحولات المنطقية، والأنواع التعدادية، والثوابت المسماة، والمصفوفات. يصف القسم الأخير في هذا الفصل كيفيّة إنشاء أنواعك الخاصة.

يُغطي هذا الفصل أساسيات استكشاف الأخطاء وإصلاحها لأنواع المعطيات الأساسية. إذا اشتملت معرفتك على أساسيات قواعد المعطيات، انتقل إلى نهاية الفصل، وراجع قائمة التحقق لتجنّب المشاكل، وانتقل إلى مناقشة أنواع المعطيات غير العادية في الفصل 13.

1.12 الأعداد بشكل عام

فيما يلي بعض التوجيهات الإرشادية لجعل استخدامك للأعداد أقل عرضة للأخطاء:

تجنّب "الأعداد السحرية"[1]. إن الأعداد السحرية هي أعداد حرفية، مثل 100 أو 47524، تظهر في منصف البرنامج بدون أي تفسير. إذا كنت تُبرمج في لغة تدعم الثوابت المسماة، فاستخدم هذه الثوابت بدلًا من الأعداد السحرية. أما إذا لم تكن هناك إمكانية لاستخدام الثوابت المسماة، فاستخدم المتغيرات العامة عندما يكون من الممكن القيام بذلك.

يؤدي تجنب الأرقام السحرية إلى تحقيق ثلاث مزايا:

- إمكانية القيام بالتغييرات بشكل أكثر موثوقية. إذا كنت تستخدم الثوابت المسماة، فلن تنظر عندها إلى واحدة من هذه المئات 100، ولن ترتكب خطأ بتغيير واحدة من هذه المئات 100، التي تُشير إلى شيء أخر.
- إمكانية القيام بالتغييرات بشكل أكثر سهولة. عندما يتغير العدد الأعظم للمُدخلات من 100 إلى 200، فإذا كنت تستخدم الأعداد السحرية فعليك إيجاد كل الأعداد 100 وتغيرهم إلى 200. وإذا كنت تستخدم 100+1 أو 100-1، فعليك أيضًا إيجاد كل الأعداد 101 و99 وتغيرهم إلى الأعداد 201 و199. أما إذا كنت تستخدم الثوابت المسماة، فببساطة تستطيع تغيير تعريف الثابت من 100 إلى 200 في مكان واحد فقط.
- تُصبح الشفرة أكثر قابلية للقراءة. تأكّد من ذلك في التعبير التالي

```
for i = 0 to 99 do..
```

يمكنك أن تحذر أن العدد 99 يُشير إلى العدد الأعظمي للمُدخلات. ولكن التعبير التالي

```
for i = 0 to MAX_ENTRIES-1 do...
```

يزيل أي شك. حتى لو كنت متأكد أن هذا العدد لن يتغير أبدًا، فإنك ستستفيد من قابلية القراءة في حالة استخدام الثابت المُسمى.

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن استخدام الثوابت المُسماة بدلًا من الأعداد السحرية، انظر القسم 7.12 "الثوابت المُسماة" في هذا الفصل.

استخدم الأعداد الحرفية 0 و1 إذا كانت هناك حاجة لذلك. يتم استخدام الأعداد 0 و1 للزيادة والإنقاص وللبدء بالحلقات عند العنصر الأول من مصفوفة. إن العدد 0 في

```
for i = 0 to CONSTANT do...
```

على مايرام. والعدد 1 في

```
total = total + 1
```

أيضاً على ما يرام. والقاعدة الجيدة هي أن القيم الحرفية الوحيدة التي يجب أن تنوجد في جسم برنامج هي الأعداد 0 و1. ولكن يجب استبدال أية أعداد حرفية أخرى مع شيء أكثر وصفية.

استبق أخطاء التقسيم على صفر. في كل مرة تستخدم فيها إشارة التقسيم (/) في معظم لغات البرمجة، فكّر فيما إذا كانت هناك إمكانية أن يكون المقام في التعبير يساوي الصفر. إذا كانت توجد مثل هكذا إمكانية، اكتب الشفرة لمنع حدوث خطأ التقسيم على صفر.

اجعل تحويلات النوع واضحة. تأكد من أن القارئ لشفرتك سينتبه إلى حدوث تحويل بين أنواع المعطيات. في سي++ يمكنك أن تكتب:

```
y = x + (float) i
```

وفي فيجوال بيسك يمكنك أن تكتب:

```
y = x + CSng( i )
```

تُساعد هذه الممارسة العملية على التأكيد أن هذا التحويل هو الذي تُريد حدوثه- تقوم المترجمات البرمجية المُختلفة بتحويلات مختلفة، لذلك فأنت تأخذ فرصتك بطريقة أخرى.

تجّنب المقارانات بين أنواع مختلفة[1]. إذا كان x عدد حقيقي ذو فاصلة عائمة وكان i عدد صحيح، فإن الاختبار

```
if ( i = x ) then...
```

هو دائماً مكفول أن يعمل. مع مرور الوقت يكتشف المترجم البرمجي أي نوع تُريد أن تستخدمه للمقارنة، ويحوّل من أحد الأنواع إلى الأنواع الأخرى، ويقوم بقليل من التقريب، ويحدّد الجواب، وعندها ستكون محظوظًا إذا اشتغل برنامجك بشكل تام. الأفضل أن تقوم بالتحويل بين الأنواع يَدَوِيًا، عندها يستطيع المترجم البرمجي أن يُقارن عددين من نفس النوع، وأنت تعرف تَمَامًا مالذي سيتم مقارنته بالضبط.

نقطة مفتاحية

اهتم بتحذيرات المترجم البرمجي. تُعلِم العديد من المترجمات البرمجية الحديثة عند استخدام أنواع عددية مختلفة في التعبير الواحد. انتبه إلى فكرة أنه في وقت ما أو آخَر يُطلب من كل مبرمج مساعدة شخص ما في تعقّب خطأ مزعج، ليكتشف أن المترجم قد حذّر من الخطأ طوال الوقت. يُصلح

[1] **إشارة مرجعية:** للحصول على تباين في هذا المثال، انظر "تجنّب مقارنات المساواة " في القسم 3.12.

المبرمجين الخبيرين شفرتهم لإلغاء كل تحذيرات المترجم البرمجي. حيث من السهل أن تدع للمترجم البرمجي القيام بعمله بدلًا عنك.

2.12 الأعداد الصحيحة integer

ضع هذه الاعتبارات في عقلك عند استخدام الأعداد الصحيحة:

تحقق من التقسيم الصحيح. عندما تستخدم الأعداد الصحيحة، فإن 7/10 لا تساوي 0.7. بل إنها بالعادة تساوي 0، أو ناقص لانهائي، أو أقرب عدد صحيح، أو – لقد فهمت الفكرة. هذا يختلف من لغة برمجة إلى لغة أخرى. وينطبق هذا أيضًا على النتائج المتوسطة. في الحياة الحقيقية، إن 10 * (10/7) = (7*10) / 10 = 7. ولكن هذا ليس تمامًا صحيح في عالم حساب الأعداد الصحيحة في البرمجة. إن 10 * (10/7) تساوي 0. إن أسهل طريقة لمعالجة هذه المشكلة هي بإعادة ترتيب التعبير البرمجي بطريقة يُصبح فيها عملية التقسيم هي الأخيرة (7*10) / 10.

تحقق من فيضان"التحميل الزائد" العدد الصحيح. عند القيام بضرب الأعداد الصحيحة أو بجمعها، فإنك تحتاج إلى أن تكون منتبه إلى أكبر عدد صحيح مسموح به. إن أكبر عدد صحيح بدون إشارة هو غالباً 2^{32}-1، وفي بعض الأحيان 2^{16}-1، أو 65535. تأتي المشكلة عندما يتم ضرب عددين صحيحين والنتيجة أكبر من العدد الصحيح الأعظمي المسموح به. على سبيل المثال، إذا ضربت العددين 250 * 300، فإن الجواب الصحيح هو 75000. ولكن إذا كان العدد الصحيح الأعظمي هو 65535، فإن جوابك سيكون تقريبًا يساوي 9464 بسبب حدوث فيضان في العدد الصحيح (75, 000 - 65, 536 = 9464). يُظهر الجدول 12-1 مجالات أنواع الأعداد الصحيحة الشائعة.

الجدول 12-1 مجالات الأنواع المختلفة من الأعداد الصحيحة

المجال	نوع العدد الصحيح
- 128 حتى 127	مع إشارة 8- بت (Signed 8-bit)
0 حتى 255	بدون إشارة 8- بت (Unsigned 8-bit)
- 32, 768 حتى 32, 767	مع إشارة 16- بت (Signed 16-bit)
0 حتى 65, 535	بدون إشارة 16- بت (Unsigned 16-bit)
- 2, 147, 483, 648 حتى 2, 147, 483, 647	مع إشارة 32- بت (Signed 32-bit)

بدون إشارة 32- بت (Unsigned 32-bit)	0 حتى 4، 294، 967، 295
مع إشارة 64- بت (Signed 64-bit)	- 9، 223، 372، 036، 854، 775، 808 حتى 9، 223، 372، 036، 854، 775، 807
بدون إشارة 64- بت (Unsigned 64-bit)	0 حتى 18، 446، 744، 073، 709، 551، 615

إن أسهل طريقة لمنع حدوث فيضان العدد الصحيح هي بالتفكير بكل عنصر من التعبير الحسابي ومحاولة تخيل أكبر قيمة يمكن لكل منها أن يفترضها. على سبيل المثال، إذا كان لدينا تعبير الأعداد الصحيحة m = j * k، فإن أكبر قيمة متوقعة للعدد j هو 200، وأكبر قيمة متوقعة للعدد k هي 25، وبذلك فإن أكبر قيمة من الممكن أن تتوقعها للنتيجة m هي 200 * 25 = 5، 000. هذا على ما يرام على الألة 32-بت حيث فيها أكبر عدد صحيح هو 2، 147، 483، 647. من ناحية أخرى، إذا كانت أكبر قيمة متوقعة للعدد j هي 200، 000 وأكبر قيمة متوقعة للعدد k هي 100، 000، فإن أكبر قيمة متوقعة للنتيجة m هي

200، 000 *100، 000 =20، 000، 000، 000.

وهذا ليس مناسبا حيث أن العدد 20، 000، 000، 000 أكبر من 2، 147، 483، 647. في هذه الحالة عليك أن تستخدم الأعداد الصحيحة 64-بت أو الأعداد الحقيقية ذات الفاصلة العائمة لموافقة القيمة الأعظمية المتوقعة للعدد m.

ضع في اعتبارك أيضًا الإضافات المستقبلية للبرنامج. إذا كان العدد **m** لن يكون أبدًا أكبر من 5، 000، فهذا عظيم. أما إذا توقعت أن ينمو العدد **m** بإطراد لعدة سنوات، فخذ هذا بعين الاعتبار.

تحقق من الفيضان في النتائج المتوسطة. إن العدد في نهاية عملية المساواة ليس العدد الوحيد الذي عليك أن تقلق حوله. افترض ان لديك الشفرة التالية:

مثال بلغة البرمجة جافا عن فيضان النتائج في الوسط

```
int termA = 1000000;
int termB = 1000000;
int product = termA * termB / 1000000;
System.out.println( "( " + termA + " * " + termB + " ) / 1000000 = " +
product );
```

إذا كنت تفكر أن تعيين الناتج هو كما يلي

(1000 ،00 ،*1000 ،000 ،) / 1000 ،000 ،

فمن الممكن أن تتوقع أن يكون الناتج 1، 000، 000. ولكن على الشفرة أن تحسب النتيجة في الوسط للعملية 1، 000، 000*1، 000، 000 000 قبل أن تتمكن من تقسيم النتيجة في النهاية على 1، 000، 000، وهذا يعني أنك تحتاج إلى عدد كبير بمقدار 1، 000، 000، 000، 000. احذر ماذا. النتيجة ستكون:

(1000000 * 1000000) / 1000000 = -727

إذا كان لديك أكبر عدد صحيح ممكن هو 2، 147، 483، 647، فإن النتيجة في وسط التعبير هي كبيرة جدًا لنوع المعطيات الصحيح. في هذه الحالة، فإن النتيجة للعملية في الوسط، التي من المفترض ان تساوي 1، 000، 000، 000، 000، ستساوي في الحقيقة - 727، 379، 968، وستحصل على نتيجة نهائية تساوي -727 بدلًامن الجواب الصحيح 1، 000، 000.

يمكنك أن تعالج مشكلة الفيضان في النتائج المتوسطة بالطريقة نفسها التي تُعالج فيها فيضان العدد الصحيح، بالتبديل إلى النوع الصحيح الطويل long-integer أو إلى النوع الحقيقي ذو الفاصلة العائمة.

3.12 الأعداد الحقيقية "ذات الفاصلة العائمة" Floating-Point



إن الاعتبار الأساسي في استخدام أعداد الفاصلة العائمة هو أنه لا يمكن تمثيل العديد من أعداد القسم العشري بشكل دقيق باستخدام الأصفار والوحدان المُتاحة على الحاسوب الرقمي. حيث يمكن عادةً تمثيل الكسور العشرية غير المحولة مثل 1/3 أو 1/7 بدقة من 7 أو 15 رقماً. في نسختي من فيجول بيسيك، تمثيل العدد الحقيقي ذو الفاصلة العائمة ذو 32-بت للكسر 1/3 يساوي 0.33333330. إنه دقيق بـ 7 أرقام بعد الفاصلة العائمة. إن هذه الدقة كافية لمعظم الأغراض، ولكنها غير دقيقة لخداعيك في بعض الأحيان.

فيما يلي بعض الإرشادات التوجيهية المتخصصة لاستخدام الأعداد ذو الفاصلة العائمة:

تجنّب عمليات الجمع والطرح للأعداد التي لديها مقادير مختلفة إلى حد كبير.[1] باستخدام متحول حقيقي ذو فاصلة عائمة من 32- بت، من الممكن أن تُنتج العملية 1، 000، 000.00 + 0.1 جواب 1، 000، 000.00، لأن 32-بت لا تعطيك أرقامًا كافية لتشمل المجال بين 1، 000، 000 و0.1. بطريقة مماثلة على الأرجح تُنتج العملية 5، 000، 000.02 - 5، 000، 000.01 العدد 0.0.

[1] **إشارة مرجعية**: من أجل كُتب الخوارزميات التي تصف طرق حل هذه المشاكل، انظر "مصادر إضافية لأنواع المعطيات" في القسم 10.1.

ما هو الحل؟ إذا كان عليك جمع سلسلة من الأرقام التي تحوي اختلافات كبيرة، عليك تصنيف الأرقام أولًا، ومن ثم جمعهم باستخدام القيم الأصغر. بطريقة مماثلة، إذا كنت تحتاج إلى جمع سلسلة لا نهائية، ابدأ من العناصر الأصغر- بشكل أساسي اجمع العناصر إلى الوراء من الأصغر. لا يُلغي هذا مشاكل التقريب، ولكن يقلّل عددها. لدى العديد من كتب الخوارزميات اقتراحات عن التعامل مع حالات مثل هذه.

تجنّب مقارانات المساواة[1]. إن الأعداد الحقيقية ذات الفاصلة العائمة التي من المفروض أن تكون متساوية ليست متساوية دائمًا. إن المشكلة الرئيسية هو أنه لا تُنتج عادةً المسارات المختلفة إلى نفس العدد الرقم نفسه. على سبيل المثال، إن `0.1` المضافة 10 مرات نادراً ما تساوي `1.0`. يُظهر المثال التالي متغيرين، nominal وsum، اللذن من المفروض أن يكونان متساويين ولكنهم غير متساويين.

مثال بلغة البرمجة جافا عن مقارنة سيئة للأعداد ذات الفاصلة العائمة

```
double nominal = 1.0;
double sum = 0.0;
for ( int i = 0; i < 10; i++ ) {
 sum += 0.1;
}
if ( nominal == sum ) {
 System.out.println( "Numbers are the same." );
}
else {
 System.out.println( "Numbers are different." );
}
```

إن المتغير nominal هو متغير من النوع الحقيقي من 64 – بت.

إن المتغير sum يساوي `10*0.1`. من المفروض أن يساوي `1.0`.

هنا المقارنة السيئة.

كما يمكنك أن تَحذُر على الأغلب، خرج هذا البرنامج هو:

```
Numbers are different.
```

تبدو قيم sum لكل سطر من تنفيذ الحلقة for كما يلي:

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

[1] 1 تساوي 2 للقيم الكبيرة بما فيه الكفاية بالنسبة للعدد 1- مجهول

وهكذا، إنها لفكرة جيدة إيجاد بديل لاستخدام مقارنة المساواة للأعداد ذات الفاصلة العائمة. إحدى الطرق الفعّالة هي بتحديد مجال الدقة المقبول ومن ثم استخدام تابع منطقي لتحديد فيما إذا كانت القيم قريبة من بعضها بما فيه الكفاية. عادةً، من الممكن أن تكتب تابع ()Equals يُعيد القيمة true إذا كانت القيم قريبة من بعضها بما فيه الكفاية، وإلا يُعيد القيمة false. في جافا، من الممكن أن يبدو هكذا تابع كما يلي:

مثال بلغة البرمجة جافا عن مقارنة الأعداد ذات الفاصلة العائمة 1

```
final double ACCEPTABLE_DELTA = 0.00001;
boolean Equals( double Term1 ،double Term2 ) {
 if ( Math.abs( Term1 - Term2 ) < ACCEPTABLE_DELTA ) {
 return true;
 }
 else {
 return false;
 }
}
```

إذا تم تحويل الشفرة في مثال" المقارنة السيئة للأعداد ذات الفاصلة العائمة"، بطريقة يمكن فيها استخدام هذه الإجرائية للمقارنات، فستبدو المقارنة الجديدة كما يلي:

```
if ( Equals( Nominal ،Sum ) )...
```

سيكون خرج البرنامج في حال استخدامه لهذا الاختبار:

```
Numbers are the same.
```

بالاستناد إلى متطلبات تطبيقك، من الممكن أن يكون من غير المناسب استخدام القيمة الحرفية للمتغير ACCEPTABLE_DELTA. من الممكن أن تضطر إلى حساب ACCEPTABLE_DELTA بالاعتماد على حجم العددين اللذين تتم مقارنتهما.

استبق أخطاء التقريب. لا تختلف مشاكل أخطاء التقريب عن مشكلة الأعداد ذات المقادير المُختلفة بشكل كبير. إنها القضية نفسها، وتُساعد العديد من نفس التقنيات في حل مشاكل التقريب. بالإضافة، فيما يلي حلول محددة شائعة لمشاكل التقريب:

- غيّر إلى نوع المتحول الذي لديه دقة أكبر. إذا كنت تستخدم نوع العدد الحقيقي ذو الفاصلة العائمة أُحادية الدقة، فغيره إلى الفاصلة العائمة مُضاعفة الدقة، وهكذا.

[1] **إشارة مرجعية:** هذا المثال هو إثبات عن المثل القائل بأن لكل قاعدة استثناء. لدى المتغيرات في هذا المثال الحقيقي أرقام في أسماءها. من أجل الاطلاع على القاعدة التي هي ضد استخدام الأرقام في اسماء المتغيرات، انظر القسم 7.11 "أنواع الاسماء التي يجب تجنبها".

- غيّر إلى المتغيرات العشرية ثنائية التشفير (binary coded decimal -BCD).[1] عادةً يأخذ المخطط العشري ثنائي التشـفير مسـاحة تخزين أكبر، ولكنه يمنع العديد من أخطاء التقريب. لهذا قيمة خاصـة إذا كانت المتحولات التي تستخدمها تُمثل الدولارات والسنتات أو أيّة كميات أخرى، التي يجب أن تتوازن على وجه التحديد.
- غيّر من متغيرات الفاصـلة العائمة إلى المتغيرات الصـحيحة. هذه هي طريقتك الخاصـة عن طريقة المتغيرات العشرية ثنائية التشفير. عليك غالباً أن تستخدم الأعداد الصحيحة 64 – بت لتحصل على الدقة التي ترغب بها. تتطلب هذه التقنية منك أن تحافظ بنفسـك على ملاحقة الجزء الكسـري من الأعداد. افترض أنك قمت بالأ صل بملاحقة الدولارات با ستخدام الأعداد ذات الفا صلة العائمة، باعتبار السـنتات هي الأجزاء الكسـرية من الدولارات. هذه عبارة عن طريقة طبيعية للتعامل مع الدولارات والسنتات. عندما تُبدل إلى الأعداد الصحيحة، فعليك ان تلاحق السنتات باستخدام الأعداد الصحيحة والدولارات باسـتخدام مضـاعفات المئة سـنت. بكلمات أخرى، تضـرب الدولارات بالعدد 100 وتحافظ على السـنتات في مجال المتغير من 0 إلى 99. قد يبدو هذا سـخيفًا للوهلة الأولى، ولكنه عبارة عن حل فعّال من حيث السـرعة والدقة. يمكنك أن تجعل عملية التلاعب هذه أسـهل وذلك بإنشـاء صـف DollarsAndCents، يُخبئ تمثيل العدد الصحيح ويدعم العمليات العددية الضرورية.

تحقق من دعم اللغة والمكتبة لأنواع معطيات محددة. لدى بعض لغات البرمجة، متضمنة فيجوال بيسك، أنواع معطيات مثل Currency، التي تدعم بشكل خاص المعطيات الحساسة إلى أخطاء التقريب.

إذا كان لدى لغتك البرمجية نوع بيانات ضمني، يؤمن مثل هكذا وظيفة، فاستخدمه.

4.12 المحارف والسلاسل المحرفية

يزود هذا القسم بعض النصائح عن استخدام السلاسل المحرفية. تُطبق النصيحة الأولى على كل السلاسل المحرفية في كل اللغات البرمجية.

تجنّب المحارف والسلاسل المحرفية السحرية.[2] إن المحارف السحرية هي المحارف الحرفية (مثل "A")، والسلاسل المحرفية هي السلاسل الحرفية (مثل "Gigamatic Accounting Program ") التي تظهر في

[1] **إشارة مرجعية**: عادة يكون تأثير الأداء للتحويل إلى التمثيل العشري الثنائي، قليل. إذا كنت قلقًا حول تأثير الاداء، انظر القسم 6.25 "ملخص منهج ضبط الشفرة ".

[2] **إشارة مرجعية**: إن قضايا استخدام المحارف والسلاسل المحرفية السحرية هي مشابهة لتلك في حالة استخدام الارقام السحرية في القسم 1.12 "الأعداد بشكل عام".

برنامج. إذا كنت تُبرمج في لغة تدعم الثوابت المُسماة، فاستخدمها بدلًاعن المحارف والسلاسل المحرفية. وإلا، فاستخدم المتغيرات العامة. يوجد العديد من الأسباب لتجنب استخدام السلاسل المحرفية الحرفية:

- من أجل السلاسل المحرفية كثيرة الحدوث مثل اسم برنامجك، وأسماء الأوامر، وإلى أخره، قد تحتاج في مرحلة من المراحل إلى تغيير محتوى السلاسل المحرفية. على سبيل المثال، من الممكن أن تتغير "Gigamatic Accounting Program" إلى "New and Improved! Gigamatic Accounting Program" في النسخ اللاحقة.
- تصبح الأسواق العالمية مهمه بشكل مُتزايد، ومن السهل ترجمة السلاسل المحرفية المُجمعة في ملف مصدر للسلاسل المحرفية، من ترجمتها في مكانها الأصلي في برنامج.
- تميل السلاسل المحرفية الحرفية إلى أخذ مساحة تخزين كبيرة. حيث يتم استخدام السلاسل المحرفية للقوائم والرسائل، وشاشات المساعدة، وصيغ الدخل، وهكذا. وإذا كان لديك العديد منهم، فإنهم ينمون بدون تحكم ويسببون مشاكل في الذاكرة. لا يتم الاهتمام بذاكرة السلاسل المحرفية في العديد من البيئات، ولكن في برمجة الأنظمة المضمنة والتطبيقات الأخرى، التي فيها مساحة التخزين أعلى من القيمة العادية، فإن حلول مشاكل مساحة تخزين السلاسل المحرفية سهلة التنفيذ إذا كانت السلاسل المحرفية مستقلة بشكل نسبي عن شفرة المصدر.
- القيم الحرفية للمحارف والسلاسل المحرفية مُبهمة. توضح التعليقات والثوابت المُسماة نواياك. في المثال التالي، إن معنى x1B0 غير واضح. يجعل استخدام الثابت ESCAPE المعنى أكثر وضوحاً.

أمثلة سي ++ لمقارنة استخدام السلاسل المحرفية

```
if ( input_char == 0x1B )...        ← سيء!
if ( input_char == ESCAPE )...      ← أفضل!
```

راقب أخطاء المقدار الواحد"أخطاء بعيداً عن واحدة" (off-by-one errors).[1] لأنه يتم فهرسة السلاسل المحرفية الفرعية كما يتم فهرسة المصفوفات، لذلك يجب مراقبة أخطاء المقدار الواحد، التي تؤدي إلى القراءة والكتابة بعد نهاية السلسلة المحرفية.

اعرف كيف تدعم لغة البرمجة والبيئة لديك المحارف القياسية يونيكود (Unicode).[2] في بعض لغات البرمجة مثل جافا كل السلاسل المحرفية هي محارف قياسية يونيكود. في بعضها الأخر مثل سي وسي++، يتطلب معالجة سلاسل المحارف القياسية يونيكود المجموعة الخاصة بها من التوابع. غالباً يُطلب التحويل بين

[1] **مصطلحات** "خطأ بعيداً من واحدة" (off-by-one error) او خطأ المقدار الواحد هو النوع المنتشر بكثرة من الأخطاء البرمجية التي تحدث عندما تخطأ بشيء واحد فقط، مثل البدء بمصفوفة من 1 بدلاً من 0، فتكون أخطأت بكل شيء في ذات السياق.

[2] cc2e.com/1285

المحارف القياسية والأنواع الأخرى من المحارف وذلك للتواصل مع المكتبات القياسية - الطرف الثالث. إذا كانت بعض السلاسل المحرفية ليست بترميز المحارف القياسية (مثل في سي وسي++)، قرر في وقت مُبكر حول ما إذا كنت تريد استخدام مجموعة المحارف القياسية على الإطلاق. إذا قررت أن تستخدم سلاسل المحارف القياسية، قرر متى وأين ستستخدمهم.

اتخذ قرار بشأن استراتيجية التدويل / الموقع في وقت مبكر من عمر البرنامج. إن القضايا المتعلقة بالتدويل / الموقع، هي قضايا رئيسية. إن الاعتبارات الرئيسية هي باتخاذ قرار بين تخزين كل السلاسل المحرفية في مورد خارجي وبين إنشاء مباني منفصلة لكل لغة أو بتحديد لغة محددة في وقت التشغيل.

إذا كنت تعرف بإنك ستحتاج فقط إلى دعم لغة أبجدية واحدة، خُذ بعين الاعتبار استخدام مجموعة المحارف ISO 8859. من أجل التطبيقات التي تدعم لغة أبجدية واحدة (مثل اللغة الإنكليزية)،[1] والتي لا تحتاج إلى دعم لغات متعددة أو لغة ايديوغرافية (مثل لغة الكتابة الصينية)، فإنه يمثل المعيار الموسّع ISO 8859 extended-ASCII بديلاً جيداً عن المحارف القياسية يونيكود.

إذا كنت تحتاج إلى دعم عدة لغات، فاستخدم المحارف القياسية يونيكود. يؤمن يونيكود دعم شامل لمجموعة المحارف الدولية أكثر من المعيار ISO 8859 أو المعايير الأخرى.

حدّد إستراتيجية تحويل متناسقة بين أنواع السلاسل المحرفية. إذا كنت تستخدم أنواع سلاسل متعددة، فإن أحد النهج الشائعة التي تساعد على المحافظة على تمييز أنواع السلاسل هي بالمحافظة على كل السلاسل في صيغة واحدة داخل البرنامج وتحويل هذه السلاسل إلى صيغ أخرى أقرب ما يمكن إلى عمليات الدخل والخرج.

السلاسل المحرفية في سي

سبب استبعاد صف السلسلة المحرفية في مكتبة القوالب القياسية للغة البرمجة سي++ معظم المشاكل التقليدية مع السلاسل المحرفية في سي. من أجل المبرمجين الذين يعملون بشكل مباشر مع السلاسل المحرفية للغة البرمجة سي، فيما يلي بعض الطرق لتجنب العثرات الشائعة:

انبته للاختلاف بين مؤشرات السلاسل المحرفية ومصفوفات المحارف. تظهر المشكلة مع مؤشرات السلاسل المحرفية ومصفوفات المحارف بسبب الطريقة التي تُعالج فيها سي السلاسل المحرفية. كُن حذر للاختلاف بينهم بطريقتين:

[1] cc2e.com/1292

- كُن حذر من أية تعبير يحوي سلسلة محرفية تتضمن إشارة يساوي. إن العمليات على السلاسل المحرفية في سي تقريباً يتم تنفيذها باستخدام strcmp()، strcpy()، strlen()، والإجرائيات الأخرى ذات الصلة. تُشير إشارات التساوي غالباً إلى نوع من خطأ المؤشر. في سي، لا تقوم عمليات الإسناد بنسخ محارف السلسلة المحرفية إلى متغير سلسلة. افترض لديك عبارة برمجية كالتالية

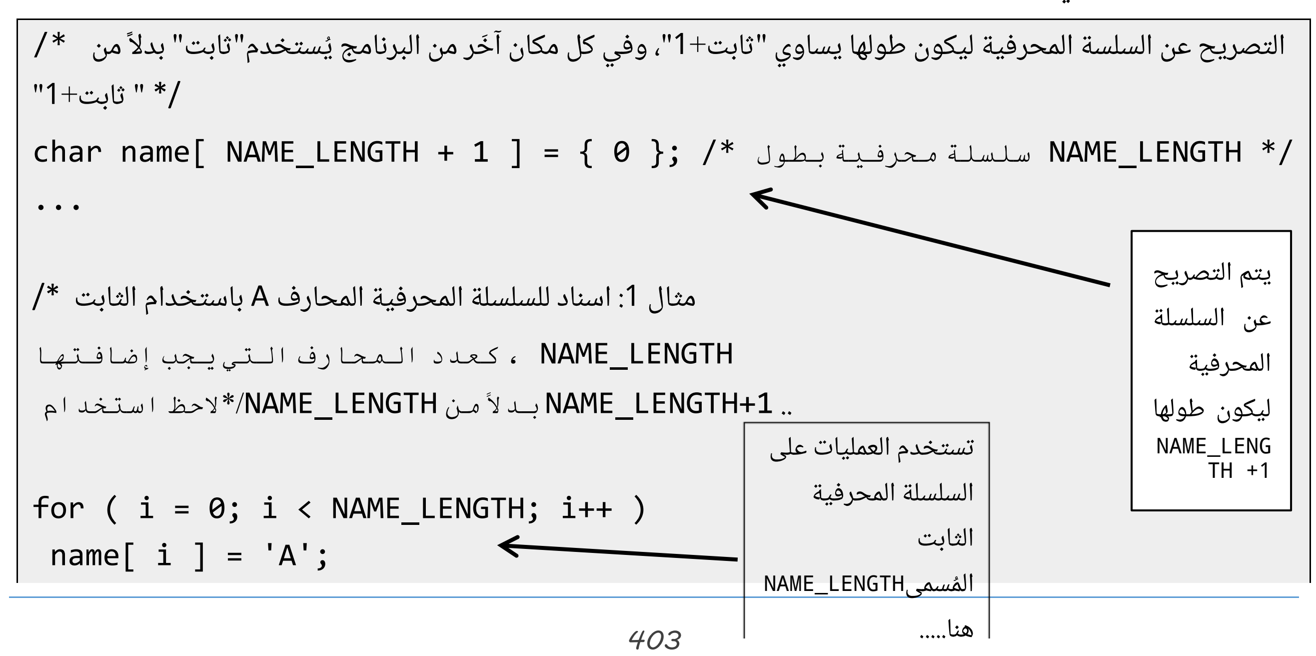
  StringPtr = "Some Text String";

في هذه الحالة، إن " Some Text String " هي مؤشر إلى سلسلة نصية حرفية، وتضع عملية الإسناد ببساطة المؤشر StringPtr إلى نقطة من هذه السلسلة النصية. حيث لا تنسخ عملية الإسناد هذه المحتويات إلى StringPtr.

- استخدم اتفاقية تسميّة لتحديد فيما إذا كانت المتغيرات هي مصفوفات محارف أو مؤشرات إلى سلاسل محرفية. إحدى الاتفاقيات الشائعة هي باستخدام ps كبادئة لتحديد مؤشر إلى سلسلة محرفية، واستخدام ach كبادئة إلى مصفوفة محارف. على الرغم من أنها ليست خاطئة دائمًا، إلا أنه يجب عليك مراعاة التعابير التي تنطوي على كلا البادئتان ps وach مع الاشتباه.

صرّح عن السلاسل المحرفية بأسلوب سي لتملك طول يساوي CONSTANT+1. في سي وسي++، إن أخطاء المقدار الواحد شائعة مع السلاسل المحرفية بأسلوب سي، لأنه من السهل نسيان أن سلسلة ذات طول n تتطلب n + 1 بايت للتخزين، ومن السهل نسيان إبقاء مجال للفاصل null (تعيين البايت إلى 0 في نهاية السلسلة). إن الطريقة الفعّالة لتجنّب مثل هكذا مشاكل هي باستخدام الثوابت المُسماة للتصريح عن السلاسل المحرفية. الشيء الأساسي في هذه الطريقة هو استخدام الثابت المُسمّى بنفس الطريقة في كل مرة. صرّح عن السلسلة المحرفية بحيث يكون طولها CONSTANT+1، ومن ثم استخدم CONSTANT للإشارة إلى طول السلسلة في بقية الشفرة. فيما يلي مثال عن هذا:

مثال بلغة البرمجة سي عن تصريحات جيدة لسلسلة محرفية



```
/* التصريح عن السلسلة المحرفية ليكون طولها يساوي "ثابت+1"، وفي كل مكان آخَر من البرنامج يُستخدم"ثابت" بدلاً من
"ثابت+1" */
char name[ NAME_LENGTH + 1 ] = { 0 }; /* سلسلة محرفية بطول NAME_LENGTH */
...

/* مثال 1: اسناد للسلسلة المحرفية المحارف A باستخدام الثابت
NAME_LENGTH ، كعدد المحارف التي يجب إضافتها
.. لاحظ استخدام NAME_LENGTH بدلاً من NAME_LENGTH+1 */

for ( i = 0; i < NAME_LENGTH; i++ )
 name[ i ] = 'A';
```

```
...
/* مثال 2: نسخ سلسلة محرفية أخرى في السلسلة المحرفية الأولى باستخدام الثابت كطول أعظمي يمكن نسخه. */

strncpy( name ،some_other_name ،NAME_LENGTH );
```

....و هنا

إذا لم تكن لديك اتفاقية للتعامل مع هذا، فإنك في بعض الأحيان ستصرح عن سلسلة محرفية ليكون طولها NAME_LENGTH وسيكون لديك العمليات عليها مع طول NAME_ LENGTH-1. وفي أوقات أخرى ستصرح عن السلسلة المحرفية ليكون طولها NAME_LENGTH+1 وسيكون لديك العمليات عليها تعمل بطول NAME_LENGTH. في كل مرة تستخدم سلسلة محرفية، عليك أن تتذكر بأية طريقة قمت بالتصريح عنها.

أما عندما تستخدم السلاسل المحرفية بنفس الطريقة في كل مرة، فلن يكون هناك حاجة لتذكر طريقة تعاملك مع كل سلسلة بشكل مستقل، وستتخلص من الأخطاء المُسببة نتيجة نسيان تفاصيل التعامل مع السلسلة المفردة. حيث يُقلل استخدام اتفاقية التسميّة هذه من التفكير الزائد والأخطاء البرمجية.

هيّئ المتغيرات إلى null **لتجنّب السلاسل المحرفية غير المنتهية**[1]. تُحدّد لغة البرمجة سي نهاية السلسلة المحرفية بإيجاد الفاصل null (بايت بالقيمة 0 في نهاية السليلة المحرفية). لا يوجد مشكلة ما هو طول السلسلة المحرفية، إن سي لا تجد نهاية السلسلة حتى تصل إلى البايت 0. إذا نسيت أن تضع null في نهاية السلسلة، من الممكن أن تعمل العمليات على السلسلة بالطريقة المتوقع منها أن تعمل بها.

يمكنك تجنب السلاسل المحرفية غير المنتهية بطريقتين. الأولى، هيّئ مصفوفات المحارف إلى القيمة 0 عندما تقوم بالتصريح عنهم:

مثال بلغة البرمجة سي عن تصريح جيد لمصفوفة محارف

```
char EventName[ MAX_NAME_LENGTH + 1 ] = { 0 };
```

ثانياً، عندما تقوم بتخصيص السلاسل المحرفية بشكل ديناميكي، هيأهم للقيمة 0 باستخدام calloc() بدلًا من malloc(). حيث تحجز calloc() ذاكرة وتهيأها إلى صفر. وتحجز malloc() ذاكرة بدون تهيأتها، أي تأخذ فرصتك بالتهيأه عند استخدام الذاكرة المخصصة من malloc()

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن تهيئة المعطيات، انظر القسم 10.3، "الإرشادات التوجيهية لتهيئة المتغيرات."

استخدم مصفوفات المحارف بدلًا من المؤشرات في لغة البرمجة سي.[1] إذا لم تكن الذاكرة مُقيدة- وغالباً هي ليست كذلك- صرّح عن كل متغيرات السلاسل المحرفية كمصفوفات من المحارف. يُساعد هذا على تجنّب مشاكل المؤشرات، وسيعطيك عندها المترجم البرمجي المزيد من التحذيرات عندما تقوم بشيء ما خاطئ.

استخدم strncpy()بدلًامن () strcpyلتجنّب السلاسل المحرفية غير المنتهية. تأتي إجرائيات السلاسل المحرفية في سي بإصدارات آمنة وأخرى خطرة. تستمر الإجرائيات الأكثر خطورة مثل strcpy() وstrcmp() بالعمل حتى تصل إلى الفاصل null. بينما تأخذ نسخهم الأمنة، strncpy() وstrncmp()، وسيط للطول الأعظمي، بحيث إذا استمرت المحارف إلى الأبد، فإن استدعاءات تابعك لن تعمل.

5.12 المتغيرات المنطقية (البوليانية) Boolean

من الصعب ارتكاب خطأ في استخدام المتغيرات المنطقية أو البوليانية، ويجعل استخدامهم بشكل مدروس برنامجك أنظف.

استخدم المتغيرات المنطقية لتوثيق برنامجك[2]. بدلًا من فقط اختبار التعابير المنطقية، يمكنك أن تسند التعبير إلى متغير يجعل تنفيذ الاختبار عديم الأخطاء. على سبيل المثال، في المقطع التالي، من غير الواضح فيما إذا كان الغرض من اختبار if هو التحقق من انتهاء حالة خطأ، أو من إتمام شيء ما أخر:

مثال بلغة البرمجة جافا لاختبار منطقي، فيه الغرض غير واضح3

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||
 ( elementIndex == lastElementIndex )
 ) {
 ...
}
```

في المقطع التالي، يجعل استخدام المتغير المنطقي الغرض من اختبار if أوضح:

مثال بلغة البرمجة جافا عن اختبار منطقي، فيه الغرض واضح

```
finished = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) );
repeatedEntry = ( elementIndex == lastElementIndex );
if ( finished || repeatedEntry ) {
 ...
}
```

[1] **إشارة مرجعية:** للمزيد من المناقشة حول المصفوفات، اقرأ القسم 8.12، "المصفوفات"، في هذا الفصل.

[2] **إشارة مرجعية:** لمزيد من التفاصيل حول استخدام التعليقات لتوثيق برنامجك، انظر الفصل 32، "التوثيق الذاتي للشفرة".

[3] **إشارة مرجعية:** للاطلاع على مثال عن استخدام التابع المنطقي لتوثيق البرنامج، انظر"جعل التعابير المعقدة بسيطة" في القسم 1.19.

استخدم المتغيرات المنطقية لتبسيط الاختبار المعقّد. غالبًا، عندما عليك أن تكتب شفرة لاختبار معقد، تقوم بعدة محاولات للحصول على الاختبار بشكل صحيح. وعندما تحاول لاحقًا تعديل الاختبار، قد يكون من الصعب فهم ما كان يقوم به الاختبار في المرة الأولى. يمكن أن تبسّط المتغيرات المنطقية الاختبار. في المثال السابق، يتم اختبار البرنامج بالفعل لشرطين: فيما إذا انتهت الإجرائية، وفيما إذا كان يتم العمل على إدخال متكرر. حيث بإنشاء المتغيرات المنطقية finished وrepeatedEntry، يُصبح اختبار if أبسط: أسهل في القراءة، وأقل تعرض للأخطاء، وأسهل في التعديل.

فيما يلي مثال آخَر عن اختبار مُعقّد:

مثال بلغة البرمجة فيجوال بيسك عن اختبار معقد

شفرة مرعبة

```
If ( ( document.AtEndOfStream() ) And ( Not inputError ) ) And _
 ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
 ( Not ErrorProcessing() ) Then
 ' do something or other
 ...
End If
```

الاختبار في المثال مُعقّد إلى حد ما ولكن ليس بشكل غير شائع. ويضع عبئًا ثقيلًا على القارئ. أتوقع أنك لن تحاول أبدًا فهم اختبار if، ولكنك ستنظر عليه وستقول: "سأفكر فيه فيما بعد إذا كنت بحاجة إليه بشكل حقيقي". انتبه إلى هذا التفكير، لأنه هو الشيء نفسه الذي يقوم به الأشخاص الأخرين عندما يقرؤون شفرتك، الحاوية على اختبارات مثل هذا الاختبار.

فيما يلي إعادة كتابة للشفرة باستخدام متغيرات منطقية مُضافة لتبسيط الاختبار:

مثال بلغة البرمجة فيجوال بيسك عن اختبار مُبسّط

```
allDataRead = ( document.AtEndOfStream() ) And ( Not inputError )
legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <=
MAX_LINES )
If ( allDataRead ) And ( legalLineCount ) And ( Not ErrorProcessing() )
Then
 ' do something or other
 ...
End If
```

هنا الاختبار المُبسّط

إن النسخة الثانية من الشفرة أبسط. أتوقع أنك ستقرأ التعابير المنطقية في اختبار if بدون أية صعوبة.

أنشئ النوع المنطقي الخاص بك. إذا كانت هناك ضرورة لذلك. لدى بعض لغات البرمجة، مثل سي++، وجافا، وفيجوال بيسك، نوع منطقي مُعرّف مسبقًا. وليس لدى البعض الأخر من لغات البرمجة، مثل سي، هذا النوع.

مثال بلغة البرمجة سي عن تعريف النوع المنطقي باستخدام typedef

```
typedef int BOOLEAN;
```

أو يمكنك القيام بهذا بهذه الطريقة، التي تُؤمن الفائدة الإضافية من تعريف true وfalse بنفس الوقت:

مثال بلغة البرمجة سي عن تعريف النوع المنطقي باستخدام Enum

```
enum Boolean {
 True=1 ،
 False=(!True)
};
```

إن التصريح عن المتغيرات باستخدام BOOLEAN بدلًامن int، يجعل استخدامهم المقصود أكثر وضوحاً ويجعل برنامجك أكثر بقليل ذاتي التوثيق.

6.12 الأنواع التعدادية

إن النوع التعدادي (enum) هو نوع من المعطيات التي تسمح لكل صف من الكائنات أن يتم وصفه باللغة الإنكليزية. يتم عادةً استخدام الأنواع التعدادية المُتاحة في سي++ وفيجوال بيسك، عند معرفتك بكل القيم المتاحة للمتغير وتُريد ان تُعبّر عنهم باستخدام كلمات باللغة الإنكليزية. فيما يلي بعض الأمثلة عن الأنواع التعدادية في فيجوال بيسك:

مثال بلغة البرمجة فيجوال بيسك عن الأنواع التعدادية

```
Public Enum Color
 Color_Red
 Color_Green
 Color_Blue
End Enum
Public Enum Country
 Country_China
 Country_England
 Country_France
 Country_Germany
 Country_India
 Country_Japan
 Country_Usa
```

```
End Enum
Public Enum Output
 Output_Screen
 Output_Printer
 Output_File
End Enum
```

إن الأنواع التعدادية هي بديل فعّال عن المخططات التي تقول فيها بصراحة: "1 هي تمثيل لـ red، و2 هي تمثيل لـ green، و3 هي تمثيل لـ blue... ". تقترح هذه الإمكانية العديد من الإرشادات التوجيهية لاستخدم الأنواع التعدادية:

استخدم الأنواع التعدادية من أجل زيادة قابلية القراءة. حيث بدلًا من كتابة عبارات برمجية مثل

```
if chosenColor = 1
```

يمكنك أن تستخدم تعابير أكثر قابلية على القراءة مثل

```
if chosenColor = Color_Red
```

في كل مرة ترى فيها رقم حرفي، اسأل نفسك هل يوجد فائدة باستبدال هذا الرقم لنوع تعدادي.

إن الأنواع التعدادية مفيدة بشكل أساسي لتعريف وسطاء إجرائية. من يعرف ماهي وسطاء التابع لهذا الاستدعاء؟



أمثلة بلغة البرمجة سي++ عن استدعاء إجرائية سيكون أفضل باستخدم الأنواع التعدادية

```
int result = RetrievePayrollData( data ،true ،false ،false ،true );
```

بالمقابل، إن الوسطاء لاستدعاء التابع هذا أكثر قابلية على الفهم:

أمثلة بلغة البرمجة سي++ عن استدعاء إجرائية، يستخدم الأنوع التعدادية لزيادة قابلية القراءة.

```
int result = RetrievePayrollData(
 data ،
 EmploymentStatus_CurrentEmployee ،
 PayrollType_Salaried ،
 SavingsPlan_NoDeduction ،
 MedicalCoverage_IncludeDependents
);
```

استخدم الأنواع التعدادية لزيادة الموثوقية. في بعض لغات البرمجة (Ada بشكل خاص)، تُتيح الأنواع التعدادية للمترجم البرمجي إجراء فحص أكثر دقة من حالة استخدام القيم الصحيحة والثوابت. حيث عند استخدام الثوابت المُسماة، ليس لدى المترجم البرمجي أية طريقة لمعرفة أن القيم المُتاحة هي فقط Color_Red، Color_Green، Color_Blue. حيث لن يعترض المترجم البرمجي على عبارات برمجية مثل

```
color = Country_England
```

أو

```
country = Output_Printer
```

أما إذا كنت تستخدم نوع تعدادي، يُصرح عن متغير مثل Color، فسيسمح المترجم البرمجي بأن يتم اسناد للمتغير فقط القيم Color_Red، Color_Green، Color_Blue.

استخدم الأنواع التعدادية لزيادة قابلية التعديل. تجعل الأنواع التعدادية شفرتك سهلة التعديل. مثلاً إذا لاحظت العيب في المخطط "1 هي تمثيل لـ red، و2 هي تمثيل لـ green، و3 هي تمثيل لـ blue... "، فسوف يكون عليك العودة إلى شفرتك وتغيير كل الأرقام 1 و2 و3 وهكذا.أما إذا كنت تستخدم نوع تعدادي، فيمكنك الاستمرار بإضافة عناصر إلى القائمة، فقط بوضعهم داخل تعريف النوع وإعادة الترجمة البرمجية للشفرة.

استخدم الأنواع التعدادية كبديل للمتغيرات المنطقية. غالبًا، لا تكون المتغيرات المنطقية كافية للتعبير عن المعاني التي تحتاج التعبير عنها. على سبيل المثال، افترض لديك إجرائية تُعيد القيمة true إذا قامت بشكل صحيح بأداء مهمتها وإلا فإنها تُعيد القيمة False. فيما بعد من الممكن أن تجد أنك بحاجة بالحقيقة إلى نوعين من False. يعني النوع الأول أن المهمة فشلت والتأثيرات محجوزة فقط بالإجرائية نفسها؛ ويعني النوع الثاني أن المهمة فشلت وسببت خطأ فادح يحتاج إلى نشره إلى بقية البرنامج. في هذه الحالة، سيكون النوع التعدادي بالقيم Status_Success، Status_Warning، Status_FatalError، أكثر فائدة من النوع المنطقي بالقيم true وfalse. ويمكن بسهولة توسيع هذا المخطط للتعامل مع أوجه الاختلاف الإضافية في أنواع النجاح أو الفشل.

تحقّق من وجود قيم غير صالحة. عندما تقوم باختبار نوع تعددي في عبارة case أو عبارة if، تحقق من وجود قيم غير صالحة. استخدم عبارة else في عبارة case لاحتواء القيمة غير الصالحة:

مثال بلغة البرمجة فيجوال بيسك عن التحقق من وجود قيم غير صالحة في النوع التعدادي.

```
Select Case screenColor
 Case Color_Red
 ...
 Case Color_Blue
 ...
 Case Color_Green
 ...
 Case Else
 DisplayInternalError( False ،"Internal Error 752: Invalid color." )
End Select
```

هنا الاختبار للقيم غير الصالحة.

عرّف المدخلات الأولى والأخيرة من تعداد لاستخدامها كحدود للحلقة. إن تعريف العناصر الأولى والأخيرة من تعداد ليكون Color_First، Color_Last، Country_First، Country_Last، وهكذا، يسمح لك بكتابة حلقة تدور بين عناصر التعداد. أنك تعد النوع التعدادي باستخدام القيم الصريحة، كما هو موضح في المثال التالي:

مثال بلغة البرمجة فيجوال بيسك عن إعداد القيم الأولى والأخيرة في نوع تعدادي.

```
Public Enum Country
 Country_First = 0
 Country_China = 0
 Country_England = 1
 Country_France = 2
 Country_Germany = 3
 Country_India = 4
 Country_Japan = 5
 Country_Usa = 6
 Country_Last = 6
End Enum
```

الآن من الممكن استخدام القيم Country_First وCountry_Last كحدود للحلقة:

مثال جيد بلغة البرمجة فيجوال بيسك عن الدوران خلال عناصر النوع التعدادي.

```
' compute currency conversions from US currency to target currency
Dim usaCurrencyConversionRate( Country_Last ) As Single
Dim iCountry As Country
For iCountry = Country_First To Country_Last
 usaCurrencyConversionRate( iCountry ) = ConversionRate( Country_Usa ،
iCountry )
Next
```

احجز المدخل الأول من التعداد كقيمة غير صالحة. عندما تقوم بالتصريح عن نوع تعدادي، احجز القيمة الأولى كقيمة غير صالحة. تسند العديد من المترجمات البرمجية العنصر الأول من النوع التعدادي للقيمة 0. إن التصريح عن عنصر مربوط إلى القيمة 0 ليكون قيمة غير صالحة، يُساعد على التقاط المتغيرات غير المهيأة بالشكل المناسب، لأنها أكثر احتمالًا أن تكون 0 من أية قيمة أخرى غير صالحة.

فيما يلي توضيح عن كيف سيبدو تصريح Country باستخدام هذه الطريقة:

أمثلة بلغة البرمجة فيجوال بيسك عن التصريح عن القيمة الأولى في تعداد ليكون قيمة غير صالحة.

```
Public Enum Country
```

```
 Country_InvalidFirst = 0
 Country_First = 1
 Country_China = 1
 Country_England = 2
 Country_France = 3
 Country_Germany = 4
 Country_India = 5
 Country_Japan = 6
 Country_Usa = 7
 Country_Last = 7
End Enum
```

حدّد بالضبط كيف يتم استخدام العناصر الأولى والأخيرة في معيار كتابة الشفرة للبرنامج، واستخدمهم باتساق. يجعل استخدام العناصر **InvalidFirst**، **First**و **Last** في التعداد، تصريحات المصفوفة والحلقات أكثر قابلية للقراءة. ولكن يُنشأ هذا الاستخدام ارتباك حول فيما إذا كانت المدخلات المتاحة في التعداد تبدأ بالعدد 0 أو 1، وفيما إذا كانت العناصر الأولى والأخيرة في التعداد صالحة. إذ تم استخدام هذه التقنية، فيجب أن يتطلب معيار كتابة الشفرة للبرنامج استخدام العناصر **InvalidFirst**، **First**و **Last** بشكل مُتّسق في كل التعدادات، وذلك لتقليل الأخطاء.

احذر من مطبات إسناد القيم الصريحة إلى عناصر تعداد. تسمح بعض لغات البرمجة بإسناد قيم محددة إلى العناصر داخل تعداد، كما هو موضّح في مثال سي++ هذا:

مثال بلغة البرمجة سي++ عن اسناد قيم صريح لتعداد.

```
enum Color {
 Color_InvalidFirst = 0،
 Color_First = 1،
 Color_Red = 1،
 Color_Green = 2،
 Color_Blue = 4،
 Color_Black = 8،
 Color_Last = 8
};
```

في هذا المثال، إذا صرّحت عن فهرس حلقة من النوع لون Color وحاولت أن تتنقل خلال هذه الألوان، فسيكون بإمكانك التنقل خلال القيم غير الصالحة 3،5،6،7، كما يمكنك ان تتنقل بين القيم الصالحة 1،2،4،8.

إذا لم يكن لدى لغتك البرمجية الأنواع التعدادية

إذا لم يكن لدى لغتك البرمجية الأنواع التعدادية، يمكنك أن تحاكيهم باستخدام المتغيرات العامة أو الصفوف. على سبيل المثال، يمكنك استخدام هذه التصريحات في جافا:

مثال بلغة البرمجة جافا عن محاكاة الأنواع التعدادية 1

```
// Country إعداد النوع التعدادي
class Country {
 private Country() {}
 public static final Country China = new Country();
 public static final Country England = new Country();
 public static final Country France = new Country();
 public static final Country Germany = new Country();
 public static final Country India = new Country();
 public static final Country Japan = new Country();
}
// Output إعداد النوع التعدادي
class Output {
 private Output() {}
 public static final Output Screen = new Output();
 public static final Output Printer = new Output();
 public static final Output File = new Output();
}
```

تجعل هذه الأنواع التعدادية برنامجك أكثر قابلية للقراءة، لأنه بإمكانك استخدام عناصر الصف العامة مثل Country.England وOutput.Screen بدلًامن الثوابت المُسماة. هذه الطريقة الخاصة في إنشاء الأنواع التعدادية هي أيضا آمنة؛ لأنه يتم التصريح عن كل نوع كصف، وسيتحقق المترجم البرمجي من الاسنادات غير المتاحة مثل Output output = Country.England (بلوخ 2001).

في لغات البرمجة التي لا تدعم الصفوف، يمكنك إنجاز التأثير نفسه من خلال الاستخدام المنضبط للمتغيرات العامة لكل عنصر من عناصر التعداد.

7.12 الثوابت المسماة

[1] **إشارة مرجعية**: في هذا الوقت الذي أكتب فيه هذا الكتاب، لا تدعم جافا الأنواع التعدادية. ولكن في وقت قراءتك لهذا الكتاب، من الممكن أنها ستدعم. وهذا مثال جيد عن "تموج موجة التكنولوجيا" المُناقشة في القسم 3.4 ،"موقعك في موجة التكنولوجيا".

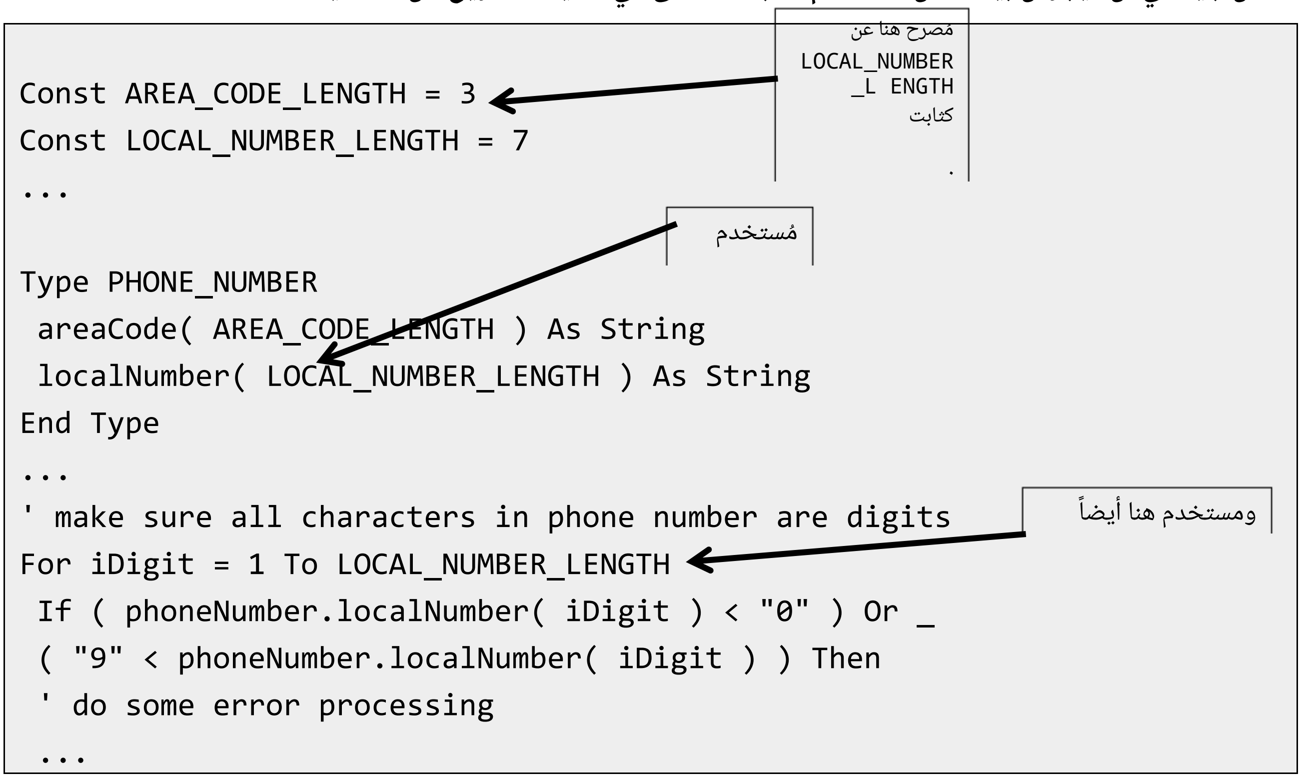
الثابت المُسمى هو مثل متغير مع استثناء بأنه لا يمكنك تغيير قيمة الثابت حالما قمت بإسنادها للثابت. تمكنك الثوابت المُسماة من الإشارة إلى كميات ثابتة، مثل العدد الأعظمي للموظفين، عن طرق استخدام اسم بدلًا من استخدام رقم، مثلًا MAXIMUM_EMPLOYEES بدلًامن 1000، على سبيل المثال.

إن استخدام الثوابت المُسماة هو طريقة لتحويل برنامجك إلى وسطاء"محددات"- أي وضع جزء من برنامجك الذي من الممكن أن يتغير في وسيط، يمكنك تغيره في مكان واحد فقط بدلًامن القيام بالعديد من التغييرات على طول البرنامج. إذا قمت بأحد المرات بالتصريح عن مصفوفة كبيرة بقدر ما تظن أنها ستحتاج أن تكون عليه كبيرة، ومن ثم نَفِذَت المساحة عندك لأن المصفوفة لم تكن كافية بما فيه الكفاية، عندها يمكنك فهم قيمة الثوابت المُسماة.

عندما يتغير حجم مصفوفة، يمكنك عندها تغيير فقط تعريف الثابت المُستخدم للتصريح عن المصفوفة. يقطع هذا "التحكم في نقطة واحدة" شوطًا كبيرًا باتجاه جعل البرمجيات "لينة": سهلة العمل معها وتغيرها.

استخدم الثوابت المُسماة في التصريحات عن المعطيات. يُساعد استخدام الثوابت المُسماة في البرنامج على زيادة قابلية القراءة وقابلية التعديل في عمليات التصريحات عن المعطيات وفي العبارات البرمجية التي تحتاج إلى أن تعرف حجم المعطيات التي تعمل معها. في المثال التالي، تستخدم LOCAL_NUMBER_LENGTH للتصريح عن طول أرقام هواتف الموظف بدلًامن استخدام الرقم الحرفي 7.

مثال جيد في ال فيجوال بيسك عن استخدام الثابت المُسمى في عملية التصريح عن المعطيات.



```
Const AREA_CODE_LENGTH = 3
Const LOCAL_NUMBER_LENGTH = 7
...

Type PHONE_NUMBER
 areaCode( AREA_CODE_LENGTH ) As String
 localNumber( LOCAL_NUMBER_LENGTH ) As String
End Type
...
' make sure all characters in phone number are digits
For iDigit = 1 To LOCAL_NUMBER_LENGTH
 If ( phoneNumber.localNumber( iDigit ) < "0" ) Or _
 ( "9" < phoneNumber.localNumber( iDigit ) ) Then
 ' do some error processing
 ...
```

هذا مثال بسيط، يمكنك على الأرجح أن تتخيل برنامج، ستحتاج فيه معلومات عن طول رقم الهاتف للموظف في عدة أماكن.

في وقت إنشاء البرنامج، يعيش كل الموظفين في بلد واحد، لذلك تحتاج فقط إلى سبع أرقام لأرقام هواتفهم. ولكن مع توسع الشركة وفروعها في عدة بلدان، ستحتاج إلى أرقام هواتف أطول. وإذا كنت قد قمت بوضع وسيط لطول رقم الهاتف، عندها يمكنك إجراء التغيير في مكان واحد فقط: في تعريف الثابت المُسمى LOCAL_NUMBER_LENGTH.

كما تتوقع، تَبيّنْ أن استخدام الثوابت المُسماة يُساعد بشكل كبير في صيانة البرنامج.[1] كقاعدة عامة، أية تقنية لتمركز التحكم على الأشياء التي من الممكن أن تتغير هي تقنية جيدة في إنقاص أعباء الصيانة (كلاس 1991).

تجنّب القيم الحرفية، حتى "الآمنة" منها. في الحلقة التالية، ماذا تفكر حول ما يمثله العدد 12؟

مثال بلغة البرمجة فيجوال بيسك عن شفرة غير واضحة

```
For i = 1 To 12
 profit( i ) = revenue( i ) - expense( i )
Next
```

بسبب الطبيعة الخاصة للشفرة، يظهر كأن الحلقة في الشفرة من المرجح أنها تدور 12 شهر في السنة. ولكن هل أن متأكد؟ هل تراهن على ذلك؟

في هذه الحالة، لا تحتاج إلى استخدام ثابت مُسمى لدعم المرونة في المُستقبل: حيث من غير المُرجح أن يتغير عدد الأشهر في العام في أي وقت. ولكن إذا كانت تترك الطريقة التي يتم فيها كتابة الشفرة أي شكل من الشك حول الغرض من الشفرة، فمن الأفضل توضيح هذا الغرض باستخدام ثابت مُسمى بشكل جيد، كما يلي:

مثال بلغة البرمجة فيجوال بيسك عن شفرة أوضح

```
For i = 1 To NUM_MONTHS_IN_YEAR
 profit( i ) = revenue( i ) - expense( i )
Next
```

هذه أفضل، ولكن لإتمام المثال، يجب إعادة تسمية فهرس الحلقة بأسم أوضَح:

مثال بلغة البرمجة فيجوال بيسك عن شفرة أكثر وضوحاً

```
For month = 1 To NUM_MONTHS_IN_YEAR
 profit( month ) = revenue( month ) - expense( month )
Next
```

يبدو هذا المثال جيد جدًا، ولكن من الممكن أيضًا تحسينه باستخدام النوع التعدادي:

1 **قراءة متعمقة:** لمزيد من التفاصيل عن قيمة التحكم في نقطة واحدة، انظر الصفحات 57-60 من صراع البرمجيات (كلاس 1991).

مثال بلغة البرمجة فيجوال بيسك عن شفرة أكثر أكثر وضوحاً

```
For month = Month_January To Month_December
 profit( month ) = revenue( month ) - expense( month )
Next
```

في المثال الأخير، لا يمكن أن يكون هناك شك في الغرض من الحلقة. حتى إذا كنت تعتقد أن العنصر الحرفي آمن، فاستخدم ثوابت مُسماة بدلًا من ذلك. كُن متعصباً حول استئصال القيم الحرفية من شفرتك. استخدم محرر نصي للبحث عن القيم 2▯3▯4▯5▯6▯7▯8▯9 والتأكد من أنك لم تستخدمهم من غير قصد.

حاكي الثوابت المُسماة باستخدام المتغيرات ذات النطاق المُناسب أو باستخدام الصفوف[1]. إذا لم تكن تدعم لغتك البرمجية الثوابت المُسماة، يمكنك أن تُنشيء الثوابت الخاصة بك. عند استخدامك لطريقة مُشابهة للطريقة المُقترحة في مثال جافا السابق، الذي تم فيه مُحاكاة الأنواع التعدادية، يمكنك أن تكسب العديد من فوائد الثوابت المُسماة. قواعد النطاق النموذجية: فضّل الترتيب التالي من النطاقات، النطاق المحلي، نطاق الصف، ثم النطاق العام.

استخدم الثوابت المُسماة بشكل مُتسق. من الخطر استخدام ثابت مُسمى في مكان وعنصر حرفي في مكان آخر، لتمثيل نفس الكيان. بعض الممارسات البرمجية هي تَسول للأخطاء؛ هذا مثل استدعاء رقم 800 ووجود أخطاء تسليمه إلى الباب الخاص بك. فإذا كانت هناك حاجة لتغيير قمة الثابت المُسمى، فسوف تقوم بتغيره وتعتقد أنك قمت بكل التغييرات اللازمة. بل ستقوم عندها بإلقاء نظرة إضافية على العناصر الحرفية، وسيطور برنامجك عيوبًا غامضة، وسيكون اصلاحها أصعب من التقاط الهاتف والصراخ من أجل المساعدة.

8.12 المصفوفات Arrays

إن المصفوفات هي النوع الأبسط الأكثر شيوعًا للمعطيات المُنّظمة. في بعض لغات البرمجة، إن المصفوفات هي النوع الوحيد للمعطيات المُنظمة. تحوي المصفوفة على مجموعة من العناصر من نفس النوع، والتي يتم الوصول إليها من خلال استخدام فهرس المصفوفة. فيما يلي بعض النصائح عن استخدام المصفوفات:

نقطة مفتاحية

تأكّد من أن كل فهارس المصفوفة داخل حدود المصفوفة. بطريقة أو بأخرى، سبب كل مشاكل المصفوفات هو من حقيقة أنه يمكن الوصول إلى عناصر المصفوفة بشكل عشوائي. تظهر أكثر مشكلة شائعة عندما يحاول برنامج الوصول إلى عنصر مصفوفة خارج الحدود. في بعض لغات البرمجة، يُنتج هذا خطأ؛ في بعضها الأخر، يُنتج هذا ببساطة نتائج غريبة وغير متوقعة.

1 **إشارة مرجعية**: لمزيد من التفاصيل حول محاكاة الأنواع التعددية، انظر " إذا لم يكن لدى لغتك البرمجية الأنواع التعدادية" في القسم السابق، القسم 12.6.

فكّر باستخدام الحاويات بدلًا من المصفوفات، أو فكّر في المصفوفات كهياكل متسلسلة. اقترح بعض من ألمع الناس في علوم الحاسوب أنه لا يمكن الوصول إلى المصفوفات بشكل عشوائي، بل فقط بشكل متسلسل (ميلز ولينجر 1986). حجتهم بأن الوصولات العشوائية في المصفوفات المُشابها إلى عبارات goto العشوائية في برنامج: تميل هذه الوصولات إلى أن تكون غير مُنضبطة، وأكثر عرضة للخطأ، ومن الصعب إثبات صحتها. اقترحوا استخدام المجموعات set، والمكدسات stack، والأرتال queue، التي يتم الوصول إلى عناصرها بشكل متسلسل، بدلًا من استخدام المصفوفات.

بيانات مثبتة

في تجربة صغيرة، وجد ميلز ولينجر أنه نتج عن التصاميم المُنشأة بهذه الطريقة متغيرات أقل ومراجع على المتغيرات أقل. كانت التصاميم فعّالة بشكل نسبي وأدت إلى إنتاج برمجيات عالية الموثوقية.

فكّر باستخدام المصفوفات الحاوية التي يمكن الوصول إليها بشكل متسلسل- المجموعات set، والمكدسات stack، والأرتال queue، وهكذا- كبدائل قبل أن تختار بشكل أتوماتيكي المصفوفة.

تحقق من نقاط النهاية في المصفوفات[1]. فقط لأنه من المفيد التفكير من خلال نقاط النهاية في هيكل حلقة، حيث يمكنك أن تلتقط العديد من الأخطاء من خلال تفحُص نقاط النهاية للمصفوفات. اسأل نفسك فيما إذا كانت الشفرة بشكل صحيح تصل إلى العنصر الأول من المصفوفة أو تصل بشكل خاطئ إلى العنصر قبل أو بعد العنصر الأول. ماذا عن العنصر الأخير؟ هل سترتكب الشفرة خطأ الخطوة الواحدة؟ وأخيرًا، اسأل نفسك فيما إذا كانت الشفرة تصل بشكل صحيح إلى العناصر في وسط المصفوفة؟

إذا كانت المصفوفة متعددة الأبعاد، تأكّد من استخدام عناصرها بالترتيب الصحيح. من السهل أن تكتب المصفوفة `Array[ i ][ j ]` عندما تعني `Array[ j ][ i ]`، لذلك خذ وقتك بالتحقق المضاعف من أنه تم استخدام الفهارس بالشكل الصحيح. فكّر باستخدام أسماء أكثر معنى من `i` و`j` في الحالات التي لا تتضح فيها أدوارهم على الفور.

انتبه من تبديل الفهارس بالخطأ (`index cross-talk`). إذا كنت تستخدم الحلقات المعششة، فمن السهل أن تكتب `Array[ j ][ i ]` عندما تعني المصفوفة `Array[ i ][ j ]`. يُدعى هذا تبديل فهارس الحلقة "`index cross-talk` ". تحقق من هذه المشكلة. الأفضل من ذلك، استخدم أسماء فهارس أكثر وضوحًا من `i` و`j` لجعل من الصعب ارتكاب أخطاء تبديل الفهارس في المقام الأول.

[1] **إشارة مرجعية**: إن قضايا استخدام المصفوفات والحلقات مُتشابهة ومرتبطة. لمزيد من التفاصيل عن الحلقات، انظر الفصل 16"التحكم بالحلقات".

في لغة البرمجة سي، استخدم الماكرو ()ARRAY_LENGTH للعمل مع المصفوفات. يمكنك تحقيق مرونة إضافية في عملك مع المصفوفات عن طريق تعريف الماكرو ()ARRAY_LENGTH كما يلي:

مثال بلغة البرمجة سي عن تعريف المسجل"الماكرو"()ARRAY_LENGTH

```
#define ARRAY_LENGTH( x ) (sizeof(x)/sizeof(x[0]))
```

عندما تقوم باستخدام عمليات في مصفوفة، بدلًا من استخدام ثابت مُسمى للحد الأعلى من حجم المصفوفة، استخدم الماكرو ARRAY_LENGTH(). فيما يلي مثال عن ذلك:

مثال بلغة البرمجة سي عن استخدام الماكرو ()ARRAY_LENGTH من أجل عمليات المصفوفة.

```
ConsistencyRatios[] =
 { 0.0 ،1.12 ،0.90 ،0.58 ،0.0 ،
 1.24 ،1.49 ،1.45 ،1.41 ،1.32 ،
 1.511.59 ،1.57 ،1.56 ،1.48 ، };
 ...
for ( ratioIdx = 0; ratioIdx < ARRAY_LENGTH( ConsistencyRatios );
ratioIdx++ );
 ...
```

هنا المكان الذي يتم فيه استخدام الماكرو.

هذه التقنية مفيدة بشكل جزئي للمصفوفات غير متعددة الأبعاد، كالمصفوفة في هذا المثال. حيث إذا قمت بإضافة أو طرح عناصر، ليس عليك أن تتذكر تغيير الثابت المُسمى الذي يصف حجم الذاكرة. أو بالطبع، تعمل هذه التقنية مع المصفوفات ذات الأبعاد، ولكن إذا كنت تستخدم هذه الطريقة، فلن تكون بحاجة دائمًا لإعداد ثابت مُسمى إضافي لتعريف المصفوفة.

9.12 إنشاء أنواع خاصة بك (إعطاء النوع اسم مستعار)



إن أنواع المعطيات المُعرفة من قبل المبرمج هي واحدة من أكثر الإمكانيات فعالية للغة البرمجة، التي تعطيك إمكانية توضيح فهمك للبرنامج. إنهم يحمون برنامجك ضد التغييرات غير المتوقعة، ويجعلون البرنامج أسهل على القراءة- بدون أن يُطلب منك تصميم، بناء أو اختبار صفوف جديدة. إذا كنت تستخدم سي، أو سي++، أو أية لغة برمجة أخرى تسمح للمستخدم بتشكيل أنواع المعطيات الخاصة به، فيجب عليك أن تستفيد من ميزات هذه الإمكانية.

لتقدير قوة إنشاء نوع معطيات جديد[1]، افترض أنك تكتب برنامج لتحويل الإحداثيات في النظام x، y، z إلى خطوط الطول والعرض والارتفاع. أنت تفكر أنه من الممكن أن تكون هناك حاجة للأعداد الحقيقية ذات الفاصلة العائمة والدقة المضاعفة، ولكن من الممكن أن تفضل أن تكتب برنامج بأعداد حقيقية ذات الفاصلة العائمة وحيدة الدقة حتى تكون متأكدًا تمامًا. يمكنك ان تنشأ نوع جديد بشكل خاص للإحداثيات باستخدام العبارة البرمجية typedef في سي، وسي++، أو ما يكافئها في لغة برمجية أخرى. فيما يلي مثال عن إعداد تعريف نوع في سي++:

مثال بلغة البرمجة سي++ عن إنشاء نوع

```
typedef float Coordinate; // للمتغيرات الإحداثية
```

يُصرح تعريف النوع هذا عن نوع جديد، الإحداثية، وهو وظيفياً نفس النوع الحقيقي float. لاستخدام النوع الجديد، تقوم فقط بالتصريح عن المتغيرات باستخدام النوع كما تفعل مع نوع محدد مسبقا مثل float. فيما ما يلي مثال:

مثال بلغة البرمجة سي++ عن استخدام النوع الذي قد قمت بإنشائه.

```
Routine1(... ) {
 Coordinate latitude; //بالدرجات latitude
 Coordinate longitude; //بالدرجات longitude
 Coordinate elevation; //بالأمتار عن مركز الأرض elevation
 ...
}
...
Routine2( ... ) {
 Coordinate x; // الإحداثية x بالأمتار
 Coordinate y; // الإحداثية y بالأمتار
 Coordinate z; // الإحداثية z بالأمتار
 ...
}
```

1 **إشارة مرجعية**: في عديد من الحالات، من الأفضل إنشاء صف بدلًامن إنشاء نوع معطيات بسيط. لمزيد من التفاصيل، انظر الفصل 6 "الصفوف العاملة".

في هذه الشفرة، تم التصريح عن كل من المتغيرات latitude، وlongitude، وelevation، وx، وy، وz، لتكن من النوع Coordinate.

الآن افترض أن البرنامج يتغير وجدت أنك تحتاج إلى استخدام المتغيرات مضاعفة الدقة للإحداثيات بعد كل هذا. لأنك قمت بتعريف نوع خاص للمعطيات الإحداثية، كل ما عليك القيام به الآن هو تغيير تعريف النوع. وعليك أن تغيره في مكان واحد فقط: في عبارة typedef. فيما يلي تعريف النوع المُغير:

مثال بلغة البرمجة سي++ عن تعريف نوع مُغير

```
typedef double Coordinate; // for coordinate
```

تم تغيير النوع الأصلي float إلى النوع double.

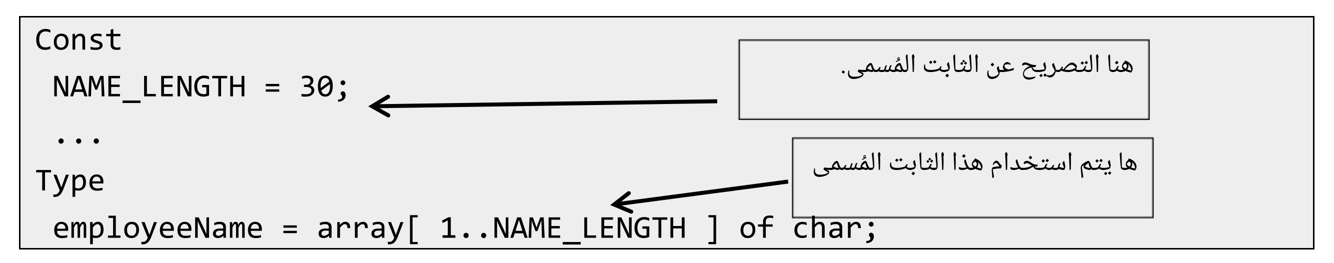
فيما يلي مثال ثاني- هذا المثال بلغة البرمجة باسكال. افترض أنك تنشأ نظام رواتب، فيه اسماء العمال بطول أعظمي يساوي 30 محرف. أخبرك المستخدمين أنه لا يوجد اسم أطول من 30 محرف. هل قمت بكتابة الرقم 30 بشكل حرفي في برنامجك؟ إذا قمت بذلك، هذا يعني أنك تثق بالمستخدمين لديك أكثر مني! طريقة أفضل هي بتعريف نوع لأسماء الموظفين.

مثال بلغة البرمجة باسكال عن إنشاء نوع لأسماء الموظفين.

```
Type
 employeeName = array[ 1..30 ] of char;
```

عندما يتم تضمين سلسلة محرفية أو مصفوفة، عادة ما يكون من الحكمة تعريف ثابت مُسمى يُشير إلى طول السلسلة المحرفية أو المصفوفة، ومن ثم استخدام الثابت المُسمى في تعريف النوع. ستجد في برنامجك العديد من الأماكن، يُستخدم فيها الثابت- هذا فقط مكان واحد ستستخدمه. فيما يلي كيف سيبدو هذا:

مثال بلغة البرمجة باسكال عن إنشاء نوع أفضل.



إن المثال الأكثر قوة هو المثال الذي يجمع فكرة إنشاء النوع الخاص بك مع فكرة إخفاء المعلومات. في بعض الحالات، إن المعلومات التي ترغب بإخفائها هي المعلومات حول نوع المعطيات.

إن مثال سي++ عن الإحداثيات هو على منتصف الطريق من إخفاء المعطيات. إذا كنت بشكل دائم تستخدم النوع Coordinate بدلًا من الأنواع float، أو double، هذا يعني أنك تُخبئ نوع المعطيات بشكل فعّال. في سي++، هذا كل ما تفعله لغة البرمجة بالنسبة لإخفاء المعلومات. بالنسبة للبقية، يجب أن يكون لدى مستخدمين شفرتك التهذيب لعدم النظر في تعريف النوع Coordinate. تُقدم سي++ قدرة إخفاء معلومات مجازية أكثر منها حرفية.

تتقدم اللغات الأخرى، مثل Ada، وتدعم الإخفاء الحرفي للمعلومات. فيما يلي موضح كيف سيبدو مقطع النوع Coordinate في رزمة لغة البرمجة Ada، التي تصرح عن هذا النوع:

مثال بلغة البرمجة Ada عن تفاصيل الإخفاء للنوع داخل رزمة.

```
package Transformation is
 type Coordinate is private;
 ...
```

تُصرح هذه العبارة عن النوع Coordinate كخاص بالنسبة للرزمة.

فيما يلي موضح كيف يبدو Coordinate في رزمة أخرى تستخدمه:

مثال بلغة البرمجة Ada عن استخدام نوع من رزمة أخرى.

```
with Transformation;
...
procedure Routine1(...) ...
 latitude: Coordinate;
 longitude: Coordinate;
begin
 -- statements using latitude and longitude
 ...
end Routine1;
```

لاحظ أنه تم التصريح عن النوع Coordinate كخاص في مواصفات الرزمة. يعني هذا أن الجزء الوحيد من البرنامج الذي يعرف تعريف النوع Coordinate هو الجزء الخاص من الرزمة Transformation. في بيئة تطوير، مع مجموعة من المبرمجين، يمكنك نشر فقط مواصفات الرزمة، التي تجعل من الصعب على مبرمج يعمل على حزمة أخرى البحث عن النوع الأساسي للإحداثيات Coordinate. عندها ستكون المعلومات بشكل حرفي مخفية. أما اللغات الأخرى مثل سي++، التي تتطلب منك نشر تعريف النوع Coordinate في ملف رأسي، إنها تخفي المعلومات بشكل مجازي.

تُوضح هذه الأمثلة الأسباب المتعددة لإنشاء الأنواع الخاصة بك:

- **لجعل عمليات التعديل أسهل.** إنه عمل قليل إنشاء نوع جديد، وهذا يعطيك الكثير من المرونة.
- **لتجنب نشر المعلومات المفرط.** تنشر الكتابة الصعبة لاشفرة تفا صيل أنواع المعطيات في برنامجك بدلًا من مركزتها في مكان واحد. هذا مثال عن مبدأ إخفاء المعلومات من أجل المركزية، المُناقش في القسم 2.6.
- **لزيادة الموثوقية.** في لغة البرمجة Ada، تستطيع تعريف أنواع مثل العمر Age في المجال بين 0- 99. ثم يولد المترجم البرمجي فحوصات في وقت التشغيل، وذلك للتحقق من أن أية متغير من النوع Age هو دائماً داخل المجال 0...99.

- **للتعويض عن نقاط ضــعف اللغة**. إذا لم يكن لدى لغتك البرمجية النوع المســبق التعريف الذي ترغب به، يمكنك إنشــاءه بنفســك. على ســبيل المثال، ليس لدى ســي النوع المنطقي أو البولياني. من ســهل تعويض هذا النقص من خلال إنشاء النوع بنفسك:

```
typedef int Boolean;
```

لماذا أمثلة إنشاء الأنواع الخاصة بك بلغتي البرمجة باسكال وAda؟

أصبحت لغات البرمجة باسكال وAda قديمة، وبشكل عام، إن اللغات التي حلّت محلها أكثر فائدة. على كل حال، في مجال تعاريف النوع البسيط، أعتقد أنه تُمثل لغات البرمجة سي++ وجافا وفيجوال بيسك حالة ثلاث خطوات إلى الأمام وخطوة واحدة إلى الوراء. تصريح بلغة البرمجة Ada كما يلي:

```
currentTemperature: INTEGER range 0..212;
```

يحوي معلومات مهمة، العبارة:

```
int temperature;
```

لا تحوي هذه المعلومات المهمة. ارجع خطوة إلى الوراء، إن التصريح عن نوع:

```
type Temperature is range 0..212;
...
currentTemperature: Temperature;
```

يسمح للمترجم البرمجي من التأكد من أنه تم اسناد currentTemperature فقط للمتغيرات الأخرى من النوع Temperature، وهناك حاجة إلى القليل جدًا من كتابة الشفرة الإضافية لتوفير هامش أمان إضافي.

بالطبع، يمكن لمبرمج أن ينشأ صف Temperature لفرض نفس الدلالات التي تم فرضها بشكل أتوماتيكي من قبل لغة البرمجة Ada، ولكن الانتقال من إنشاء نوع معطيات بسيط في سطر واحد من الشفرة إلى إنشاء صف هي خطوة كبيرة. في العديد من الحالات، يمكن للمبرمج أن ينشأ نوع بسيط، ولكن لن يصل إلى الجهد الإضافي لإنشاء صف.

إرشادات توجيهية لإنشاء الأنواع الخاصة بها

ضع هذه الإرشادات في اعتبارك عندما تقوم بإنشاء الأنواع "المعرفة عن طريق المستخدم"[1]:

أنشئ الأنواع بأسماء موجهة وظيفيًا. تجنّب أسماء النوع التي تُشير إلى نوع معطيات الحاسوب المُضمنة في النوع. استخدم أسماء النوع التي تُشير إلى أجزاء من مشكلة العالم الحقيقي التي يُمثلها النوع الجديد. في الأمثلة السابقة، أنشأت التعريفات أنواع جيدة التسمية من أجل الإحداثيات والأسماء - كيانات العالم الحقيقي.

1 **إشارة مرجعية:** في كل حالة، فكر فيما إذا كان إنشاء صف من الممكن ان يعمل أفضل من نوع المعطيات البسيط، لمزيد من التفاصيل، انظر الفصل 6 "الصفوف العاملة".

بشكل مُشابه، يمكنك إنشاء أنواع معطيات من أجل العملة، رموز الدفع، الأعمار وما إلى ذلك- جوانب مشاكل العالم الخارجي.

كُن حذر من إنشاء أسماء الأنواع التي تُشير إلى الأنواع مُسبقة التعريف. تُشير أسماء النوع مثل *BigInteger*، و*LongString* إلى معطيات حاسوب بدلًا من مشاكل العالم الخارجي. إن الحسنة الكبيرة من إنشاء نوعك الخاص هو توفير طبقة عزل بين برنامجك ولغة التنفيذ. إن أسماء النوع التي تُشير إلى أنواع لغة البرمجة الأساسية تصنع ثقوب في طبقة العزل تلك. إنهم لا يعطونك ميزة أكبر لاستخدام النوع المُعرف مُسبقاً. من ناحية أخرى، تؤمن لك الأسماء الموجهة على المشكل قابلية تعديل أسهل وتصريحات ذاتية التوثيق للمعطيات.

تجنّب الأنواع مُسبقة التعريف. إذا كان يوجد احتمال تغير النوع، تجنّب استخدام الأنواع مسبقة التعريف في أي مكان، ولكن فقط في عبارة typedef أو في تعريفات النوع. من السهل إنشاء أنواع جديدة وظيفية التوجه، ومن الصعب تغيير المعطيات في برنامج يستخدم الأنواع مسبقة التعريف. علاوة على ذلك، يوثّق استخدام تصريحات النوع الموجه وظيفياً جزئيًا المتغيرات المُصرحة بواسطتها. يُخبرك تصريح مثل Coordinate x الكثير عن x بدلًامن التصريح مثل float x. استخدم الأنواع الخاصة بك بقدر ما تستطيع.

لا تعد تعريف الأنواع مُسبقة التعريف. يولد تغيير تعريف النوع القياسي ارتباك. على سبيل المثال، إذا كان لدى لغتك البرمجية نوع مُسبق التعريف للأعداد الصحيحة Integer، فلا يجب أن تُنشأ نوع جديد خاص بك يُدعى Integer. من الممكن أن ينسى قُراء شفرتك بأنك قد أعدت تعريف النوع، ويفترضون أن النوع Integer الذي يرونه هو نفسه الـ Integer الذي اعتادوا رؤيته.

عرّف أنواع بديلة من أجل قابلية التنقل. على النقيض من النصيحة التي تقول بأنه لا يجب تغيير تعريف نوع قياسي، من الممكن أن ترغب بتعريف أنواع بديلة للأنواع القياسية، وهكذا تستطيع جعل المتغيرات على أجهزة مختلفة تمثيل نفس الكيانات بالضبط. على سبيل المثال، تستطيع تعريف نوع INT32 وتستخدمه بدلًامن int، أو النوع LONG64 لتستخدمه بدلًامن النوع long. أصلًا، من الممكن أن يكون الاختلاف الوحيد بين هذين النوعين هو الأحرف الكبيرة لاسم النوع البديل. ولكن عندما تَنقُل البرنامج إلى جهاز جديد، من الممكن إعادة تعريف النسخة ذات الأحرف الكبيرة بحيث يمكن مقابلتها لأنواع المعطيات على الجهاز الأصلي.

تأكّد من عدم تعريف الأنواع من السهل الإخطاء فيها مع الأنواع المعرّفة مسبقًا. سيكون من الممكن تعريف INT بدلًامن INT32، ولكن من الأفضل إنشاء اختلاف واضح بين النوعين، الذي عرفته والأنواع مسبقة التعريف من قبل لغة البرمجة.

خُذ بعين الاعتبار إنشاء صف بدلًا من استخدام `typedef`. ممكن أن يؤدي استخدام `typedef` إلى إخفاء المعلومات حول نوع المتغير الأساسي. في بعض الحالات، على كل حال، من الممكن أن ترغب بمرونة إضافية وتحكم يمكنك تحقيقه بواسطة إنشاء صف. لمزيد من التفاصيل انظر الفصل 6 "الصفوف العاملة".

لائحة اختبار: المعطيات الأساسية[1]

الأعداد بشكل عام

- هل تتجنب الشفرة الأعداد السحرية؟
- هل تستبق الشفرة أخطاء التقسيم على صفر؟
- هل التحويلات بين الأنواع واضحة؟
- إذا تم اســتخدام متغيرين من نوعين مختلفين في نفس التعبير البرمجي، فهل ســيتم تقييم التعبير كما كنت تنوي أن يكون عليه؟
- هل تتجنب الشفرة المقارنات بين أنواع مختلطة؟
- هل يتم ترجمة البرنامج مع تحذيرات؟

الأعداد الصحيحة

- هل تعمل التعابير التي تستخدم القسمة الصحيحة بالطريقة التي يُراد أن تعمل بها؟
- هل تتجنب التعابير الصحيحة مشاكل فيضان العدد الصحيح؟

الأعداد ذات الفاصلة العائمة

- هل تتجنب الشــفرة عمليات الجمع والطرح على الأعداد التي لديها مقادير مختلفة إلى حد كبير؟
- هل تمنع الشفرة بشكل مُنظم أخطاء التقريب؟
- هل تتجنب الشفرة مقارنة التساوي بين أعداد الفاصلة العائمة؟
- المحارف والسلاسل المحرفية
- هل تتجنب الشفرة المحارف والسلاسل المحرفية السحرية؟
- هل المراجع على الســلاســل المحرفية متحررة من أخطاء المقدار الواحد off-by-one errors؟

[1] cc2e.com/1206

إشارة مرجعية: من أجل قائمة التحقق التي تُطبَق على قضايا المعطيات العامة بدلًا من قضايا الأنواع الخاصة من المعطيات، انظر قائمة التحقق في الصفحة 257 في الفصل 10 "القضايا العامة في استخدام المتغيرات". ومن أجل قائمة التحقق من أجل اعتبارات تسمية المتغيرات، انظر قائمة التحقق في الصفحة 288 في الفصل 11 "قوة أسماء المتغير".

- هل تتعامل شفرة سي مع مؤ شرات الـسلا سل المحرفية ومـصفوفات المحارف بـشكل مختلف؟
- هل تتبع شــفرة ســي اتفاقية التصــريح عن الســلاســـل المحرفية ليكون طولها CONSTANT+1؟
- هل تـستخدم شفرة سي مـصفوفات من المحارف بدلًا من ا ستخدام المؤ شرات، عندما يوجد إمكانية لذلك؟
- هل تُهيء شفرة سي السلاسل المحرفية للقيمة NULL، وذلك لتجنب السلاسل المحرفية غير المنتهية؟
- هل تـستخدم شفرة سي ()strncpy بدلًا من ا ستخدام()strcpy؟ وتستخدم ()strncat بدلًا من استخدام ()strncmp؟
- المتغيرات المنطقية
- هل يستخدم البرنامج متغيرات منطقية إضافية لتوثيق الاختبارات الشرطية؟
- هل يستخدم البرنامج متغيرات منطقية إضافية لتسهيل الاختبارات الشرطية؟

الأنواع التعدادية

- هل يســتخدم البرنامج الأنواع التعدادية بدلًامن اســتخدام الثوابت المُســماة، من أجل قابلية القراءة المُحسنة لديها، والموثوقية المحسنة وقابلية التعديل؟
- هل يســتخدم البرنامج الأنواع التعدادية بدلًا من اســتخدام المتغيرات المنطقية، عندما لا يمكن التقاط استخدام المتغير بشكل كامل مع true وfalse.
- هل تستخدم الاختبارات الأنواع التعدادية لاختبار القيم غير الصالحة؟
- هل الإدخال الأول في النوع التعدادي محجوز للقيمة "غير صالح (invalid)"؟

الثوابت المسماة

- هل يســتخدم البرنامج الثوابت المُســماة من أجل تصــريحات المعطيات وحدود الحلقة بدلًا من الأرقام السحرية؟
- هل تم ا ستخدام الثوابت المُسماة باتساق- لا يتم ا ستخدم الثوابت المُسماة في بعض الأماكن، وقيم حرفية في أماكن أخرى؟

المصفوفات

- هل فهارس المصفوفة داخل حدود المصفوفة؟
- هل المراجع على المصفوفات متحررة من أخطاء المقدار الواحد off-by-one errors؟

- هل كل الأجزاء في المصفوفات متعددة الأبعاد هي بالترتيب الصحيح؟
- في الحلقات المتداخلة، هل تم استخدام المتغير الصحيح كفهرس، مع تجنب الأخطاء في فهارس الحلقة؟

إنشاء الأنواع

- هل يستخدم البرنامج نوع مختلف لكل صنف من المعطيات التي من الممكن أن تتغير؟
- هل تم توجيه أسماء النوع إلى كائنات العالم الحقيقي التي تُمثلها، بدلًا من أن يتم توجيها إلى أنواع لغة البرمجة؟
- هل أسماء النوع وصفية بما فيه الكفاية لتساعد على توثيق تصريحات المعطيات؟
- هل تم تجنب إعادة تعريف الأنواع مُسبقة التعريف؟

نقاط مفتاحية

- يعني العمل مع أنواع معطيات محددة تَذكُر العديد من القواعد الفردية لكل نوع. اســتخدام قائمة التحقق لهذا الفصل للتأكد من أنك قد أخذت بعين الاعتبار المشكلات الشائعة.
- يجعل توليد الأنواع الخا صة بك برامجك أ سهل على التعديل وموثّقه ذاتيًا بشكل أكبر، وهذا في حالة إذا كانت لغة البرمجة التي تستخدمها تدعم هذه الإمكانية.
- عندما تقوم بتوليد نوع بســيط باســتخدام *typedef* أو ما يكافئها، خُذ بعين الاعتبار فيما إذا كان عليك توليد صف جديد بدلًا من ذلك.

الفصل 13

أنواع البيانات غير العادية

المحتويات[1]

مواضيع ذات صلة

- أنواع البيانات الأساسية: الفصل 12
- البرمجة الوقائية: الفصل 8
- بنى التحكم غير العادية: الفصل 17
- التعقيد في تطوير البرمجيات: القسم 5-2

بعض اللغات تدعم الأنواع الغريبة من البيانات بالإضافة إلى أنواع البيانات التي نوقشت في الفصل 12، "أنواع البيانات الأساسية". ويصف القسم 13-1 متى يمكنك استخدام الهياكل بدلا من الصفوف في بعض الظروف. ويصف القسم 13-2 العناصر الداخلية والخارجية لاستخدام المؤشرات. إذا واجهتك مشاكل مرتبطة باستخدام البيانات الشاملة، فالقسم 13-3 يشرح كيفية تجنب هذه الصعوبات. إذا كنت تعتقد أن أنواع البيانات الموصوفة في هذا الفصل ليست نفسها الأنواع التي تقرأ عنها عادة في كتب البرمجة غرضية التوجه الحديثة، فأنت على حق. لهذا السبب يسمى الفصل "أنواع البيانات غير العادية".

13. 1 الهياكل

يشير مصطلح "هيكل" إلى البيانات التي تم إنشاؤها من أنواع أخرى. لأن المصفوفات هي حالة خاصة، يتم التعامل معها بشكل منفصل في الفصل 12. هذا القسم يتعامل مع هياكل البيانات المنشأة من قبل المستخدم،

[1] cc2e.com/1378

الهياكل في "سي" و"سي++"، والهياكل في مايكروسوفت فيجوال بيسيك. في جافا و"سي++"، الصفوف أيضا تؤدي أحيانا دور الهياكل (عندما يتكوّن الصف بالكامل من أعضاء البيانات العامة بدون أي إجراءات عامة).

سترغب عموما في إنشاء صفوف بدلا من الهياكل بحيث يمكنك الاستفادة من الخصوصية والوظائف التي تقدمها الصفوف بالإضافة إلى البيانات العامة التي تدعمها الهياكل. ولكن أحيانا التلاعب مباشرة بكتل من البيانات يمكن أن يكون مفيدا، لذلك تجد هنا بعضا من الأسباب لاستخدام الهياكل:

استخدام الهياكل لتوضيح علاقات البيانات الهياكل تحزم مجموعات من العناصر المرتبطة معاً. وأحيانا يكون أصعب جزء من معرفة البرنامج هو معرفة البيانات التي ترتبط مع البيانات الأخرى. إنها مثل الذهاب إلى بلدة صغيرة والسؤال "من يقرب من". ستكتشف أن الجميع أقارب نوعاً ما، ولكن ليس تماماً، وأنت لن تحصل على إجابة جيدة.

إذا تمت هيكلة البيانات بعناية، فمعرفة العناصر المرتبطة يصبح أسهل بكثير. فيما يلي مثال للبيانات التي لم تتم هيكلتها:

مثال في فيجوال بيسيك على متغيرات مضللة، وغير مهيكلة

```
name = inputName
address = inputAddress
phone = inputPhone
title = inputTitle
department = inputDepartment
bonus = inputBonus
```

لأن هذه البيانات غير مهيكلة، يبدو كما لو أن كل عبارات الإسناد ترتبط ببعضها. في الواقع، الاسم والعنوان والهاتف هي متغيرات مرتبطة بالموظفين الأفراد، والعنوان والقسم والمكافأة هي متغيرات مرتبطة بالمشرف. لا توفر قطعة الشفرة هذه أي تلميح بأن هناك نوعان من البيانات في العمل. في قطعة الشفرة أدناه، استخدام الهياكل يجعل العلاقات أكثر وضوحا:

مثال في فيجوال بيسيك على متغيرات أكثر إعلاماً وتنظيماً

```
employee.name = inputName
employee.address = inputAddress
employee.phone = inputPhone
supervisor.title = inputTitle
supervisor.department = inputDepartment
supervisor.bonus = inputBonus
```

في الشفرة التي تستخدم المتغيرات المهيكلة، من الواضح أن بعض البيانات مرتبطة مع موظف، وبيانات أخرى مرتبطة مع مشرف.

استخدام الهياكل لتبسيط العمليات على كتل من البيانات يمكنك الجمع بين العناصر المرتبطة في هيكل وتنفيذ عمليات على الهيكل. فإنه من الأسهل العمل على هيكل من تنفيذ نفس العملية على كل من العناصر. كما أنها أكثر موثوقية، وتتطلب عدداً أقل من أسطر التعليمات البرمجية.

لنفرض أنه لديك مجموعة من عناصر البيانات التي ترتبط معاً، على سبيل المثال، بيانات عن موظف في قاعدة بيانات الموظفين. إذا لم يتم دمج البيانات في بنية، مجرد نسخ مجموعة من البيانات يمكن أن تنطوي على الكثير من العبارات. وإليك مثال في فيجوال بيسيك:

مثال في فيجوال بيسيك على نسخ مجموعة من عناصر البيانات غير المصقولة

```
newName = oldName
newAddress = oldAddress
newPhone = oldPhone
newSsn = oldSsn
newGender = oldGender
newSalary = oldSalary
```

في كل مرة تريد فيها نقل معلومات عن الموظف، يجب أن تقوم بكل هذه المجموعة من العبارات. إذا أضفت قطعة جديدة من معلومات الموظف – على سبيل المثال، "numWithholdings" – عليك أن تجد كل مكان يوجد فيه كتلة من الإسنادات وإضافة إسناد لـ "newNumWithholdings = oldNumWithholdings".

تخيل كم ستكون رهيبة مبادلة البيانات بين اثنين من الموظفين. لا يجب عليك استخدام خيالك – هذه هي:



مثال في فيجوال بيسيك على مبادلة مجموعتين من البيانات – الطريق الصعب

```
//مبادلة بيانات الموظف الجديدة والقديمة
previousOldName = oldName
previousOldAddress = oldAddress
previousOldPhone = oldPhone
previousOldSsn = oldSsn
previousOldGender = oldGender
previousOldSalary = oldSalary
oldName = newName
oldAddress = newAddress
oldPhone = newPhone
oldSsn = newSsn
oldGender = newGender
oldSalary = newSalary
newName = previousOldName
newAddress = previousOldAddress
newPhone = previousOldPhone
newSsn = previousOldSsn
newGender = previousOldGender
newSalary = previousOldSalary
```

طريقة أسهل للتعامل مع المشكلة هي التصريح عن متغير مهيكل:

مثال فيجوال بيسيك على التصريح عن الهياكل

```
Structure Employee
name As String
address As String
phone As String
ssn As String
gender As String
salary As long
End Structure
Dim newEmployee As Employee
Dim oldEmployee As Employee
Dim previousOldEmployee As Employee
```

الآن يمكنك تبديل جميع العناصر في هياكل الموظف القديمة والجديدة بثلاث عبارات:

مثال فيجوال بيسيك على طريقة أسهل لمبادلة مجموعتين من البيانات

```
previousOldEmployee = oldEmployee
oldEmployee = newEmployee
newEmployee = previousOldEmployee
```

إذا كنت ترغب في إضافة حقل مثل "numWithholdings"، يمكنك ببساطة إضافته إلى تصريح الهيكل. ولا يلزم تعديل العبارات الثلاثة المذكورة أعلاه ولا العبارات المماثلة في جميع أنحاء البرنامج. "سي++" وغيرها من اللغات لديها قدرات مماثلة.

استخدام الهياكل لتبسيط قوائم الوسطاء[1] يمكنك تبسيط قوائم وسطاء الإجرائية باستخدام المتغيرات المهيكلة. هذه التقنية هي مماثلة لتلك التي عرضت للتو. بدلا من تمرير كل عنصر من العناصر المطلوبة بشكل فردي، يمكنك تجميع العناصر ذات الصلة في هيكل وتمرير الكل كبُنية مجموعة. وإليك مثال على الطريقة الصعبة لتمرير مجموعة من الوسطاء المرتبطة ببعضها:

مثال فيجوال بيسيك على استدعاء غير متقن لإجرائية دون هيكل

```
HardWayRoutine( name, address, phone, ssn, gender, salary )
```

وهنا مثال على طريقة سهلة لاستدعاء إجرائية باستخدام متغير مهيكل يحتوي على عناصر قائمة الوسطاء الأولى (في المثال السابق):

مثال على استدعاء إجرائية أنيقة مع هيكل

```
EasyWayRoutine( employee )
```

إذا كنت ترغب في إضافة "numWithholdings" إلى النوع الأول من الاستدعاء، عليك أن تخوض خلال الشفرة الخاصة بك وتغير كل استدعاء لـ " HardWayRoutine()". إذا قمت بإضافة عنصر

[1] **إشارة مرجعية** لمزيد من التفاصيل حول كمية البيانات التي يمكن مشاركتها بين الإجرائيات، راجع "حافظ على الاقتران ضعيفاً " في القسم 3-5.

"numWithholdings" إلى الموظف، لا يجب عليك أن تغير الوسطاء لـ " ()HardWayRoutine" على الإطلاق.

يمكنك أن تحمل هذه التقنية إلى أقصى الحدود، ووضع جميع المتغيرات في برنامجك في متغير واحد كبير ومن ثم تمريره الى كل مكان[1]. يتجنب المبرمجون الحريصون تجميع البيانات أكثر مما هو ضروري منطقيا. وعلاوة على ذلك، يتجنب المبرمجون الحريصون تمرير هيكل كوسيط عندما يكون هناك حاجة لحقل واحد أو اثنين فقط من الهيكل – يمررون الحقول المحددة المطلوبة بدلا من ذلك. وهذا جانب من جوانب إخفاء المعلومات: بعض المعلومات مخفيّة في الإجرائية، وبعضها مخفي عن الإجرائية. يتم تمرير المعلومات على أساس الحاجة إلى المعرفة.

استخدام الهياكل للتقليل من الصيانة لأنك تجمع البيانات المرتبطة عند استخدام الهياكل، فإن تغيير هيكل يتطلب تغييرات أقل على كامل البرنامج. هذا صحيح بشكل خاص في أقسام الشفرة التي لا ترتبط منطقيا بالتغيير في الهيكل. وبما أن التغييرات تميل إلى إحداث أخطاء، فإن تغييرات أقل يعني أخطاء أقل. إذا كان هيكل الموظف لديه حقل عنوان وقررت حذفه، فلن تحتاج إلى تغيير أي من قوائم الوسطاء أو عبارات الإسناد التي تستخدم البنية بأكملها. بالطبع، عليك تغيير أي شفرة تتعامل على وجه التحديد مع عناوين الموظفين، ولكن هذا من الناحية المفاهيمية مرتبط بحذف حقل العنوان ويصعب التغاضي عنه.

الميزة الكبيرة لهيكلة البيانات يمكن رؤيتها في أقسام الشفرة التي لا تحمل أي علاقة منطقية لحقل العنوان. في بعض الأحيان يكون للبرامج عبارات تشير بشكل مفاهيمي إلى مجموعة من البيانات بدلا من المكونات الفردية. في مثل هذه الحالات، تتم الإشارة إلى المكونات الفردية، مثل حقل العنوان، فقط لأنها جزء من المجموعة. هذه الأقسام من الشفرة ليس لديها أي سبب منطقي للعمل مع حقل العنوان على وجه التحديد، وهذه الأقسام من السهل التغاضي عنها عند تغيير العنوان. إذا كنت تستخدم هيكل، فمن الصواب التغاضي عن هذه الأقسام لأن الشفرة تشير إلى مجموعة البيانات المرتبطة بدلا من الإشارة الى كل مكون على حدة.

13.2 المؤشرات



استخدام المؤشر هو واحد من أكثر المناطق عرضة للخطأ في البرمجة الحديثة، إلى حد أن اللغات الحديثة مثل "جافا" و"سي شارب" و"فيجوال بيسيك" لا توفر نوع بيانات مؤشر. استخدام المؤشرات معقد بالأصل، واستخدامها بشكل صحيح يتطلب أن يكون لديك فهم ممتاز لمخطط إدارة المترجم

[1] **إشارة مرجعية** لمزيد من التفاصيل حول مخاطر تمرير الكثير من البيانات، راجع " حافظ على الاقتران ضعيفاً" في القسم 5-3.

للذاكرة. العديد من المشاكل الأمنية الشائعة، وخاصة تجاوزات المخزن المؤقت، يمكن أن ترجع إلى الاستخدام الخاطئ للمؤشرات (هوارد وليبلانك 2003) (Howard and LeBlanc 2003).

حتى إذا كانت لغتك لا تتطلب منك استخدام مؤشرات، الفهم الجيد للمؤشرات سوف يساعدك على فهم كيفية عمل لغة البرمجة الخاصة بك. ومن شأن جرعة سخيّة من ممارسات البرمجة الوقائية أن تساعد أكثر من ذلك.

نموذج لفهم مؤشرات

من الناحية النظرية، يتكون كل مؤشر من جزأين: موقع في الذاكرة، ومعرفة لكيفية تفسير محتويات هذا الموقع.

الموقع في الذاكرة

الموقع في الذاكرة هو عنوان، غالبا ما يتم التعبير عنه في التدوين الست عشري. عنوان على معالج 32 بت سيكون قيمة من 32 بت، مثل "0x0001EA40". يحتوي المؤشر نفسه على هذا العنوان فقط. لاستخدام البيانات التي يشير إليها المؤشر، عليك أن تذهب إلى هذا العنوان وتفسر محتويات الذاكرة في هذا الموقع. إذا تفحّصت الذاكرة في هذا الموقع، فإنها ستكون مجرد مجموعة من البتات. ويجب أن تُفسَّر ليصبح لها معنى.

معرفة كيفية تفسير المحتويات

معرفة كيفية تفسير محتويات موقع في الذاكرة يتم توفيرها من نوع قاعدة المؤشر. إذا كان المؤشر يشير إلى عدد صحيح، ما يعنيه ذلك حقا هو أن المترجم يفسر موقع الذاكرة التي يقدمها المؤشر على أنه عدد صحيح. بالطبع، يمكن أن يكون لديك مؤشر لعدد صحيح، مؤشر لسلسلة نصيّة، ومؤشر لفاصلة عائمة، جميعها تشير إلى نفس موقع الذاكرة. ولكن واحد فقط من المؤشرات يفسر محتويات هذا الموقع بشكل صحيح.

عند التفكير في المؤشرات، فإنه من المفيد أن نتذكر أن الذاكرة ليس لديها أي طريقة تفسير أصلية مرتبطة بها. وفقط من خلال استخدام نوع محدد من المؤشرات يمكن تفسير البتات في موقع معين على أنها بيانات ذات معنى.

ويبين الشكل 13-1 عدة منظورات لنفس الموقع في الذاكرة، وتفسيرها في عدة طرق مختلفة.

0A 61 62 63 64 65 66 67 68 69 6A

المنظور: محتويات الذاكرة الخام المستخدمة في الأمثلة القادمة (بالنظام الست عشري)

التفسير: لا يوجد تفسير ممكن دون متغير مؤشر مرتبط بها

0A 61 62 63 64 65 66 67 68 69 6A

المنظور: سلسلة [10] (بتنسيق فيجوال بيسيك مع بايت الطول أولا)

التفسير: abcdefghij



المنظور: عدد صحيح 2-بايت

التفسير: 24842



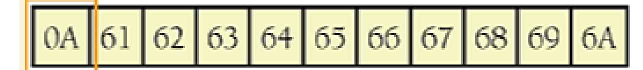
المنظور: فاصلة عائمة 4-بايت

التفسير: "E+00214.17595656202980"



المنظور: عدد صحيح 4-بايت

التفسير: 1667391754



المنظور: محرف

التفسير: حسب نوع المحرف (0A بالنظام الست عشري وفق ترميز ASCII أو 10 بالنظام العشري)

الشكل 13-1 يتم عرض مقدار الذاكرة المستخدمة من قبل كل نوع من البيانات من خلال الخطوط المزدوجة.

وفي كل حالة من الحالات الواردة في الشكل 13-1، يشير المؤشر إلى الموقع الذي يحتوي على القيمة "x0A0" بالنظام الست عشري. ويعتمد عدد البايتات المستخدمة ما بعد "A0" على كيفية تفسير الذاكرة. وتعتمد طريقة استخدام محتويات الذاكرة أيضا على كيفية تفسير الذاكرة. كما أنه يعتمد على المعالج الذي تستخدمه، لذا ضع ذلك في الاعتبار إذا حاولت تكرار هذه النتائج على Cray Desktop الخاص بك[1]. يمكن تفسير نفس محتويات

1 **مصطلحات** Cray: Cray Desktop شركة تصنع حواسيب عملاقة، وأنظمة لتخزين البيانات وتحليلها.

الذاكرة الخام على أنها سلسلة أو عدد صحيح أو فاصلة عائمة أو أي شيء آخر – كل ذلك يعتمد على النوع الأساسي للمؤشر الذي يشير إلى الذاكرة.

نصائح عامة حول المؤشرات

مع العديد من أنواع العيوب، تحديد الخطأ هو أسهل جزء من التعامل مع الخطأ، أما تصحيحه ذلك هو الجزء الصعب. أخطاء المؤشرات تختلف عن ذلك. خطأ المؤشر عادة ما يكون نتيجة مؤشر يشير لمكان لا ينبغي أن يؤشر إليه. عند إسناد قيمة إلى متغير مؤشر سيء، فأنت تقوم بكتابة بيانات إلى مساحة من الذاكرة لا ينبغي أن تكتب بها. وهذا ما يسمى "تلف الذاكرة". أحيانا تلف الذاكرة ينتج تعطل رهيب وناري للنظام؛ أحيانا يغير نتائج الحساب في جزء آخر من البرنامج؛ وأحيانا يتسبب لبرنامجك بتخطي الإجرائية بشكل غير متوقع؛ وأحيانا لا يفعل شيء على الإطلاق. في الحالة الأخيرة، خطأ المؤشر هو قنبلة موقوتة، تنتظر أن تدمر برنامجك قبل أن تعرضه – على أكثر عملائك أهمية – بخمس دقائق.

أعراض أخطاء المؤشر تميل إلى أن تكون غير مرتبطة بأسباب أخطاء المؤشر. وبالتالي، فإن معظم العمل في تصحيح خطأ المؤشر هو تحديد السبب.

يتطلب العمل مع المؤشرات بنجاح استراتيجية ذات شقين. أولا، تجنب وضع أخطاء المؤشر منذ البداية. من الصعب جدا العثور على أخطاء المؤشر لذلك فإن التدابير الوقائية الإضافية لها ما يبررها. ثانيا، اكشف أخطاء المؤشر في أقرب وقت ممكن بعد إضافتها للشفرة. أعراض أخطاء المؤشر غير منتظمة لذلك فإن التدابير الإضافية لجعل الأعراض أكثر قابلية للتنبؤ لها ما يبررها. وإليك كيفية تحقيق هذه الأهداف الرئيسية:

اعزل عمليات المؤشر في الإجرائيات أو الصفوف لنفرض أنك تستخدم لائحة مترابطة في عدة أماكن في أحد البرامج. بدلا من اجتياز القائمة يدويا كل مكان تُستخدم فيه، اكتب إجرائيات وصول مثل "NextLink()"، "PreviousLink()"، "InsertLink()"، "DeleteLink()".

من خلال التقليل من عدد الأماكن التي يتم الوصول فيها الى المؤشرات، فأنت تقلل من احتمالية قيامك بالأخطاء الطائشة التي تنتشر في جميع أنحاء برنامجك وتأخذ إلى الأبد للعثور عليها. ونظرا لأن الشفرة مستقلة نسبيا عن تفاصيل تنفيذ البيانات، يمكنك أيضا تحسين فرصة إعادة استخدامها في برامج أخرى. كتابة إجرائيات لتحصيص المؤشر هي طريقة أخرى لمركزة التحكم بالبيانات الخاصة بك.

صرح وهيّء المؤشرات في نفس الوقت تعيين قيمة أولية لمتغير بالقرب من مكان التصريح عنه هو عموما ممارسة جيدة للبرمجة، وهذا كله يكون أكثر قيمة عند العمل مع المؤشرات. هنا مثال على ما لا يجب أن تفعله:

مثال في سي++ على تهيئة سيئة لمؤشر

```
Employee *employeePtr;
// العديد من التعليمات البرمجية
...
employeePtr = new Employee;
```

حتى إذا كانت هذه الشفرة تعمل بشكل صحيح في البداية، فإنها عرضة للخطأ تحت التعديل لأن هناك فرصة أن شخص ما سيحاول استخدام "employeePtr" بين النقطة التي تم فيها التصريح عن المؤشر والوقت الذي تم فيه تهيئته. وهنا نهج أكثر أمانا:

مثال في سي++ على تهيئة جيدة لمؤشر

```
// العديد من التعليمات البرمجية
...
Employee *employeePtr = new Employee;
```

احذف المؤشرات في نفس مستوى النطاق الذي تم تحصيصها فيه حافظ على تناظر تحصيص وإلغاء تحصيص المؤشرات. إذا استخدمت مؤشر ضمن نطاق معين، استدعي "new" لتحصيص و"delete" لإلغاء تحصيص المؤشر ضمن نفس النطاق. إذا قمت بتحصيص مؤشر ضمن إجرائية، ألغِ التحصيص ضمن إجرائية شقيقة. إذا قمت بتحصيص مؤشر ضمن باني الكائن، ألغِ التحصيص ضمن هادم الكائن. الإجرائية التي تقوم بتحصيص الذاكرة ومن ثم تتوقع من شفرة العميل أن تقوم بإلغاء تحصيص الذاكرة يدوياً تنشئ تناقضاً يجهّز لحدوث خطأ.

تحقق من المؤشرات قبل استخدامها قبل استخدامك لمؤشر في جزء حرج من برنامجك، تأكد من أن يكون موقع الذاكرة الذي يشير إليه منطقياً. على سبيل المثال، إذا كنت تتوقع أن تكون مواقع الذاكرة بين "StartData" و"EndData"، يجب أن تنظر بشك إلى المؤشر الذي يشير قبل "StartData" أو بعد "EndData". سيتوجب عليك أن تحدد ما هي قيم "StartData" و"EndData" في البيئة الخاصة بك. يمكنك ضبط هذا للعمل تلقائياً إذا كنت تستخدم المؤشرات من خلال إجرائيات الوصول بدلا من التعامل معها مباشرة.

تحقق من المتغير المشار إليه بواسطة المؤشر قبل استخدامه في بعض الأحيان يمكنك إجراء عمليات تدقيق منطقية على القيمة التي يشير إليها المؤشر. على سبيل المثال، إذا كنت من المفترض أن تشير إلى قيمة عدد صحيح بين "0" و"1000"، فيجب أن تشك بالقيم التي تتجاوز "1000". إذا كنت تشير إلى سلسلة بأسلوب "سي++"، قد تشك بالسلاسل ذات أطوال أكبر من "100". ويمكن أيضا أن يتم ذلك تلقائيا إذا كنت تعمل مع المؤشرات من خلال إجرائيات الوصول.

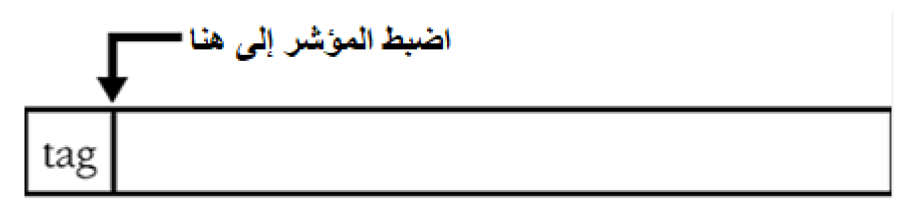
استخدم حقول "علامة التعريف" للتحقق من الذاكرة التالفة "حقل العلامة" أو "علامة التعريف" هو حقل تضيفه إلى بنية(هيكل) لغرض فحص الأخطاء فقط. عندما تقوم بتحصيص متغير، ضع قيمة والتي يجب أن تبقى دون تغيير في حقل العلامة. عند استخدامك البنية – خاصة عند حذف الذاكرة – تحقق من قيمة حقل العلامة. إذا لم يكن حقل العلامة يحتوي على القيمة المتوقعة، فقد تعرضت البيانات للتلف.

عندما تحذف المؤشر، أتلف الحقل بحيث إذا حاولت عن طريق الخطأ تحرير نفس المؤشر مرة أخرى، سوف تكشف التلف. على سبيل المثال، لنفترض أنك بحاجة إلى تحصيص 100 بايت:

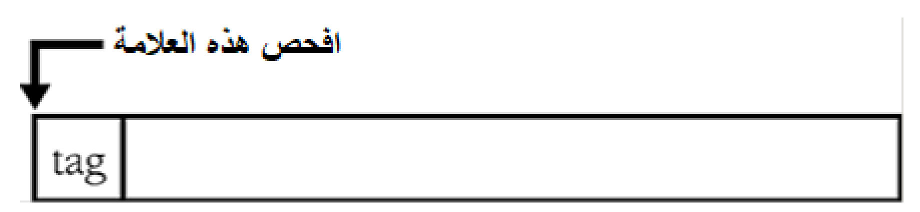
أولا، احجز 104 بايت جديدة، 4 بايت أكثر من المطلوب.

104 bytes

قم بتعيين أول 4 بايت لقيمة علامة التعريف ثم قم بإرجاع المؤشر إلى الذاكرة التي تبدأ بعد ذلك.



عندما يحين الوقت لحذف المؤشر، تحقق من العلامة.



إذا كانت العلامة صحيحة، فقم بتعيينها على "0" أو قيمة أخرى والتي تتعرف عليها أنت وبرنامجك كقيمة علامة غير صالحة. لن ترغب بأن تخطئ بالقيمة على أنها لعلامة صالحة بعد تحرير الذاكرة. عين البيانات إلى "0" أو "0xCC" أو بعض القيم الأخرى غير العشوائية لنفس السبب.

وأخيرا، احذف المؤشر.

حرر كامل الـ 104 بايت

وضع علامة التعريف في بداية كتلة الذاكرة التي قمت بتحصيصها يسمح لك بالتحقق من المحاولات الزائدة لإزالة تحصيص كتلة الذاكرة دون الحاجة إلى الاحتفاظ بقائمة بجميع كتل الذاكرة التي قمت بتحصيصها. وضع علامة التعريف في نهاية كتلة الذاكرة يسمح لك بالتحقق من الكتابة فوق الذاكرة بعد الموقع الذي كان من المفترض أن يستخدم. يمكنك استخدام العلامات في بداية ونهاية الكتلة لتحقيق كلا الهدفين.

يمكنك استخدام هذا النهج بالتنسيق مع منطقية التحقق المقترح سابقاً – التحقق من أن المؤشرات بين "StartData" و"EndData". للتأكد من أن المؤشر يشير إلى موقع معقول، بدلا من التحقق من وجود نطاق محتمل من الذاكرة، تحقق من أن المؤشر هو في قائمة المؤشرات المحصصة.

يمكنك التحقق من حقل العلامة مرة واحدة فقط قبل حذف المتغير. علامة تالفة ستخبرك أنه في وقت ما خلال حياة هذا المتغير قد تلفت محتوياته. كلما أكثرت من التحقق من حقل العلامة، مع ذلك، فإنك تقترب من جذر المشكلة وسوف تكشف عن التلف.

أضف زوائد واضحة بديل لاستخدام حقل العلامة هو استخدام حقول معينة مرتين. إذا كانت البيانات في الحقول الزائدة غير متطابقة، فستعلم أن الذاكرة قد تلفت. هذا يمكن أن يؤدي إلى الكثير من النفقات العامة إذا كنت تتعامل مع المؤشرات مباشرة. إذا قمت بعزل عمليات المؤشر في الإجرائيات، ومع ذلك، فإنه يضيف شفرة مكررة في مواضع قليلة.

استخدم متغيرات مؤشرات إضافية للوضوح بكل الوسائل، لا تبخل باستخدام متغيرات المؤشرات. تم توضيح هذه النقطة في مكان آخر على أنه لا ينبغي استخدام متغير لأكثر من غرض واحد. وينطبق هذا بشكل خاص على متغيرات المؤشرات. من الصعب معرفة ما يفعله شخص ما مع لائحة مترابطة دون الحاجة إلى معرفة سبب استخدام متغير "genericLink" واحد مرارا وتكرارا أو ما الذي يشير إليه "-pointer->next >last->next". فكر في جزء الشفرة هذا:

مثال سي++ على شفرة تقليدية لإدراج عقدة

```
void InsertLink(
    Node *currentNode,
    Node *insertNode
    ) {
//  "currentNode" بعد "insertNode" إدخال
    insertNode->next = currentNode->next;
    insertNode->previous = currentNode;
    if ( currentNode->next != NULL ) {
        currentNode->next->previous = insertNode;
    }
    currentNode->next = insertNode;
}
```

هذا السطر صعب بلا داعٍ

هذا هي الشفرة التقليدية لإدخال عقدة في لائحة مترابطة، وهو صعب الفهم بلا داعٍ. يتضمن إدخال عقدة جديدة ثلاثة أغراض: العقدة الحالية والعقدة التي تتبع العقدة الحالية حالياً والعقدة المراد إدراجها بينهما. يعبر جزء الشفرة بشكل صريح عن كائنين فقط: "insertNode" و"currentNode". إنه يجبرك على معرفة وتذكر أن "currentNode->next" هي أيضا معنيّة. إذا حاولت رسم ما يحدث دون العقدة التي تتبع "currentNode" في الأصل، سوف تحصل على شيء من هذا القبيل:

currentNode	insertNode

يوجد رسم بياني أفضل يحدد جميع الكائنات الثلاثة. وسيبدو مثل هذا:

startNode	newMiddleNode	followingNode

إليك الشفرة التي تشير بشكل صريح إلى جميع العناصر الثلاثة المعنية:

مثال في سي++ على شفرة أكثر قابلية للقراءة لإدراج عقدة

```
   void InsertLink(
        Node *startNode,
        Node *newMiddleNode
        ) {
"followingNode" و "startNode" بين "newMiddleNod" إدخال
//
        Node *followingNode = startNode->next;
        newMiddleNode->next = followingNode;
        newMiddleNode->previous = startNode;
        if ( followingNode != NULL ) {
            followingNode->previous = newMiddleNode;
        }
        startNode->next = newMiddleNode;
   }
```

تحتوي هذه الشفرة على سطر إضافي من التعليمات البرمجية، لكن بدون الجزء "currentNode->next->previous" من الشفرة الأولى، فإنها من الأسهل لمتابعتها.

بسّط تعبيرات المؤشر المعقدة من الصعب قراءة تعبيرات المؤشرات المعقدة. إذا كانت شفرتك تحتوي على تعبيرات مثل "p->q->r->s.data"، فكّر في الشخص الذي عليه أن يقرأ التعبير. وهنا مثال فظيع بشكل خاص:



مثال في سي++ على تعبير مؤشر من الصعب فهمه

```
  for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
      netRate[ rateIndex ] = baseRate[ rateIndex ] * rates->discounts-
>factors->net;
  }
```

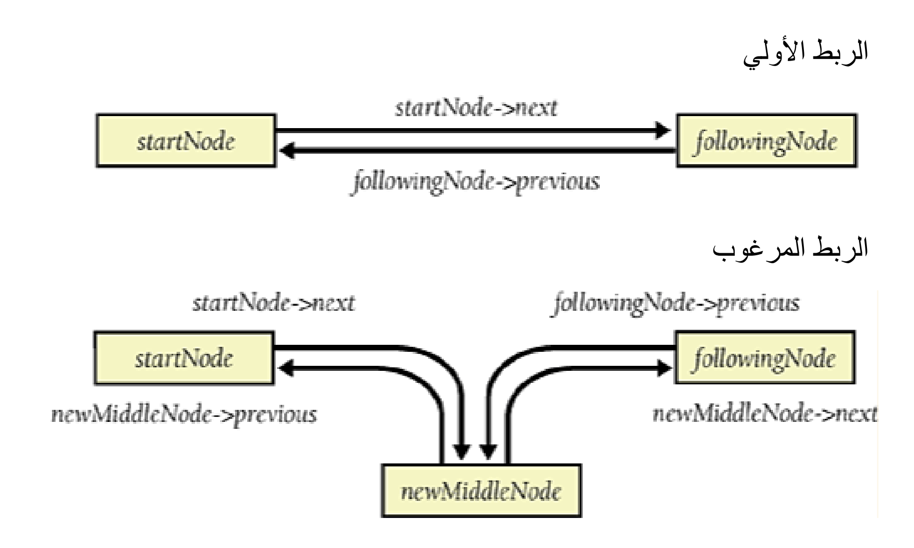
التعبيرات المعقدة مثل تعبير المؤشر في هذا المثال تجعل أن تكون الشفرة يجب أن تُفهم بدلا من أن تُقرأ. إذا كانت شفرتك تحتوي على تعبير معقد، فقم بتعيينها لمتغير مسمّى بشكل جيد لتوضيح القصد من العملية. وإليك نسخة محسنة من المثال:

مثال في سي++ على تبسيط تعبير المؤشر المعقد

```
  quantityDiscount = rates->discounts->factors->net;
  for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
      netRate[ rateIndex ] = baseRate[ rateIndex ] *
quantityDiscount;
  }
```

مع هذا التبسيط، لن تحصل فقط على مكاسب في سهولة القراءة، ولكن قد تحصل أيضا على تحسين في الأداء من تبسيط عملية المؤشر داخل الحلقة. كالمعتاد، سيكون عليك قياس فائدة الأداء قبل أن ترهن أي ورقة نقدية على ذلك.

ارسم صورة تفاصيل شفرة المؤشرات يمكن أن تصبح مربكة. وعادة ما يساعد رسم صورة. على سبيل المثال، قد تبدو الصورة لمشكلة الإدراج في اللائحة المترابطة كما هو موضح في الشكل 13-2. [1]



الشكل 13-2 مثال للصورة التي تساعدنا على التفكير في الخطوات التي تنطوي عليها مؤشرات إعادة الربط.

احذف المؤشرات في اللوائح المترابطة بالترتيب الصحيح هناك مشكلة شائعة في العمل مع اللوائح المترابطة ديناميكيا، هي أنه يتم تحرير المؤشر الأول في اللائحة أولاً حيث بعد ذلك لا يمكن الوصول إلى المؤشر التالي في اللائحة. لتجنب هذه المشكلة، تأكد من أنه لديك مؤشر إلى العنصر التالي في اللائحة قبل تحرير المؤشر الحالي.

خصص مساحة احتياطية من الذاكرة إذا كان البرنامج يستخدم الذاكرة الديناميكية، تحتاج إلى تجنب مشكلة النفاد المفاجئ للذاكرة، وترك المستخدم وبيانات المستخدم الخاص بك تضيع في مساحة ذاكرة الوصول

1 **إشارة مرجعية** يمكن أن تصبح الرسوم البيانية مثل تلك الموجودة في الشكل 13-2 جزءاً من الوثائق الخارجية للبرنامج. لمزيد من التفاصيل حول ممارسات التوثيق الجيدة، راجع الفصل 32، "قانون التوثيق الذاتي".

العشوائي. إحدى الطرق لإعطاء برنامجك هامش خطأ هو تخصيص مساحة ذاكرة. حدد مقدار الذاكرة التي يحتاجها برنامجك لحفظ العمل، والتنظيف، والخروج بأمان. خصص هذه الكمية من الذاكرة في بداية البرنامج كمساحة احتياطية، واتركها جانباً. عند نفاد الذاكرة، حرر المساحة الاحتياطية، ونظّف، وأغلق.

أتلف نفاياتك[1] من الصعب تصحيح أخطاء المؤشر لأن النقطة التي يشير إليها المؤشر في الذاكرة لتصبح غير صالحة هي غير محددة. في بعض الأحيان سوف تبدو محتويات الذاكرة صالحة بعد فترة طويلة من تحرير المؤشر. في أوقات أخرى، سوف تتغير الذاكرة على الفور.

في سي، يمكنك فرض الأخطاء التي تتعلق باستخدام المؤشرات الملغى تخصيصها لتكون أكثر اتساقا من خلال الكتابة فوق كتل الذاكرة ببيانات غير مهمة "مهملة" مباشرة قبل أن يُلغى تخصيصها. كما هو الحال مع العديد من العمليات الأخرى، يمكنك القيام بذلك تلقائيا إذا كنت تستخدم إجرائيات الوصول. في سي، في كل مرة تقوم فيها بحذف مؤشر، يمكنك استخدام شفرة مثل هذه:

مثال في "سي" على إجبار كائن ملغى تخصيصه ليحتوي على بيانات غير مهمة

```
pointer->SetContentsToGarbage();
delete pointer;
```

وبطبيعة الحال، لن تعمل هذه التقنية في سي++ حيث يشير المؤشر إلى كائن، ويتطلب منك تنفيذ إجرائية وضع المحتويات في المهملات لكل كائن.

عيّن المؤشرات إلى فارغ (null) بعد حذفها أو تحريرها نوع شائع من أخطاء المؤشر هو "المؤشر المعلق"، استخدام مؤشر تم حذفه أو تحريره. أحد أسباب صعوبة الكشف عن أخطاء المؤشر هو أن الخطأ في بعض الأحيان لا ينتج أي أعراض. من خلال تعيين المؤشرات إلى "فارغ" بعد تحريرها، لا تتغير حقيقة أنه يمكنك قراءة البيانات التي أشار إليها المؤشر المعلق. ولكن يمكنك التأكد من أن كتابة البيانات إلى المؤشر المعلق ينتج خطأ. فمن المحتمل أن يكون خطأً قبيحاً وسيئاً وكارثياً، ولكن على الأقل ستجده أنت بدلا من أن يجده شخص آخر.

يمكن توسيع الشفرة السابقة لعملية الحذف في المثال السابق للتعامل مع هذا أيضا:

مثال في سي++ على تعيين مؤشر إلى فارغ بعد حذفه

```
pointer->SetContentsToGarbage();
delete pointer;
pointer = NULL;
```

[1] **اقرأ أيضا** من أجل مناقشة ممتازة للنهج الآمن للتعامل مع المؤشرات في سي، انظر كتابة شفرة صلبة (ماغوير 1993) (Maguire 1993).

تحقق من وجود مؤشرات سيئة قبل حذف متغير أحد أفضل الطرق لتخريب البرنامج هي " `delete()`" أو "`free()`" (حذف أو تحرير) مؤشر بعد أن كان بالفعل قد تم حذفه أو تحريره. لسوء الحظ، عدد قليل من اللغات يكشف عن مشكلة من هذا النوع.

تعيين المؤشرات التي تم تحريرها إلى فارغ (`null`) يسمح لك أيضاً بالتحقق مما إذا كان المؤشر قد تم تعيينه إلى `null` قبل استخدامه أو محاولة حذفه مرة أخرى؛ إذا لم تقم بتعيين المؤشرات التي تم تحريرها إلى `null`، لن يكون لديك هذا الخيار. وهذا يقترح إضافة أخرى إلى شفرة حذف المؤشر:

مثال في سي++ على التأكد من أن المؤشر ليس Null قبل حذفه

```
  ASSERT( pointer != NULL, "Attempting to delete null pointer."
);
  pointer->SetContentsToGarbage();
  delete pointer;
  pointer = NULL;
```

تتبع تخصيصات المؤشر احتفظ بقائمة بكل المؤشرات التي قمت بتخصيصها. يسمح لك هذا بالتحقق مما إذا كان المؤشر في القائمة قبل التخلص منه. فيما يلي مثال لكيفية تعديل شفرة حذف المؤشر القياسية لتشمل ذلك:

مثال في سي++ على التحقق مما إذا كان المؤشر قد تم تخصيصه

```
 ASSERT( pointer != NULL, "Attempting to delete null pointer."
);
 if ( IsPointerInList( pointer ) ) {
     pointer->SetContentsToGarbage();
     RemovePointerFromList( pointer );
     delete pointer;
     pointer = NULL;
 }
 else {
     ASSERT( FALSE, "Attempting to delete unallocated pointer."
);
 }
```

اكتب إجرائيات لمركزة استراتيجيتك لتجنب مشاكل المؤشر كما ترون من هذا المثال، يمكن أن ينتهي بك المطاف الى الكثير من الشفرة الاضافية في كل مرة يتم انشاء مؤشر جديد أو يتم حذفه. بعض التقنيات الموضحة في هذا القسم غير متكافئة أو زائدة عن الحاجة، وأنت لا تريد أن يكون لديك استراتيجيات متعددة متعارضة قيد الاستخدام في نفس قاعدة الشفرة. على سبيل المثال، لا تحتاج إلى إنشاء قيم علامات العلامات والتحقق منها إذا كنت تحتفظ بقائمة المؤشرات الصالحة الخاصة بك.

يمكنك تقليل النفقات العامة للبرمجة وتقليل فرصة حدوث الأخطاء عن طريق إنشاء إجرائيات تغطية لعمليات المؤشر الشائعة. في سي++، يمكنك استخدام هاتين الإجرائيتين:

- **الإنشـاء الجديد الآمن** (SAFE_NEW): هذه الإجرائية تسـتدعي (new) لتحصـيص المؤشـر، وتضـيف المؤشــر الجديد إلى قائمة المؤشــرات المحصــصــة، وتعيد المؤشــر المحصــص حديثا إلى الإجرائية المسـتدعية. ويمكن أيضـا التحقق من وجود اسـتثناء أو قيمة إعادة فارغة من (new)، (ويعرف أيضـا با سم خطأ "خارج الذاكرة") في هذا المكان الوحيد فقط، مما يبسط معالجة الأخطاء في أجزاء أخرى من برنامجك.
- **الحذف الآمن** (SAFE_DELETE)**:** هذه الإجرائية تتحقق لمعرفة ما إذا كان المؤشــر الممرر لها موجود في قائمة المؤ شرات المحـصـصة. إذا كان في القائمة، فإنها تحدد المتغير الذي يـشير إليه المؤ شر إلى قيم مهملة، وتزيل المؤشر من القائمة، وتستدعي عملية الحذف في سي++ لإلغاء تحصيص المؤشر، وتعين المؤشــر إلى فارغ. إذا لم يكن المؤشــر في القائمة، فإن إجرائية الحذف الآمن (SAFE_DELETE) تعرض رسالة تشخيصية وتوقف البرنامج.

منفذه هنا كمسجل (ماكرو)، إجرائية الحذف الآمن (SAFE_DELETE) تشبه هذه:

مثال في سي++ على تغليف شفرة حذف المؤشر

```
 #define SAFE_DELETE( pointer ) { \
     ASSERT( pointer != NULL, "Attempting to delete null
pointer."); \
     if ( IsPointerInList( pointer ) ) { \
         pointer->SetContentsToGarbage();\
         RemovePointerFromList( pointer ); \
         delete pointer; \
     pointer = NULL; \
     } \
     else { \
         ASSERT( FALSE, "Attempting to delete unallocated pointer."
); \
     } \
 }
```

في سي++[1]، ستقوم هذه الإجرائية بحذف مؤشرات فردية، ولكنك ستحتاج أيضا إلى تنفيذ إجرائية "SAFE_DELETE_ARRAY" لحذف المصفوفات.

عن طريق مركزة معالجة الذاكرة في هاتين الإجرائيتين، يمكنك أيضا جعل (SAFE_NEW) و(SAFE_DELETE) تتصرف بشكل مختلف في وضع التصحيح مقابل وضع الإنتاج. على سبيل المثال، عندما تكتشف

1 **إشارة مرجعية** للحصول على تفاصيل حول التخطيط لإزالة الشفرة المستخدمة لتصحيح الأخطاء، راجع "خطة لإزالة أدوات تصحيح الأخطاء" في القسم 8-6.

(SAFE_DELETE) محاولة لتحرير مؤشر فارغ أثناء التطوير، فإنها قد توقف البرنامج، ولكن أثناء الإنتاج فهي ببساطة قد تسجل خطأ وتتابع المعالجة. اثلة لحذف المصفوفات.

يمكنك بسهولة ملائمة هذا المخطط إلى (calloc) و(free) في لغة "سي" وإلى اللغات الأخرى التي تستخدم المؤشرات.

استخدام تقنية غير المؤشرات المؤشرات أكثر صعوبة للفهم من المعتاد، فهي عرضة للخطأ، وهي تميل إلى أن تتطلب اعتماديات على الآلة، وشفرة غير محمولة. إذا كنت تستطيع التفكير في بديل لاستخدام المؤشرات يعمل بشكل معقول، وفر على نفسك بعض الصداع واستخدمها بدلا من ذلك.

مؤشرات مؤشر-سي++

"سي++" تقدم بعض الطرق المحددة المتعلقة باستخدام المؤشرات والمراجع.[1] تصف الأقسام الفرعية التالية الإرشادات التي تنطبق على استخدام المؤشرات في "سي++":

فهم الفرق بين المؤشرات والمراجع في "سي++"، كل من المؤشرات (*) والمراجع (&)، تدل بشكل غير مباشر الى كائن. بالنسبة لغير المختصين، الفرق الوحيد يبدو أنه تمييز تجميلي بحت بين الإشارة الى حقل كـ "object->field" مقابل "object.field". الاختلافات الأكثر أهمية هي أن المرجع يجب أن يشير دائما إلى كائن، في حين أن المؤشر يمكن أن يشير إلى الفراغ، وما يشير إليه المرجع لا يمكن تغييره بعد أن تتم تهيئة المرجع.

استخدم المؤشرات لوسطاء "التمرير بالمرجع" واستخدم المراجع الثابتة لوسطاء "التمرير بالقيمة" الافتراضيات في "سي++" هي تمرير الوسطاء إلى الإجرائيات بالقيمة بدلا من المرجع. عند تمرير كائن إلى إجرائية بالقيمة، تقوم "سي++" بإنشاء نسخة من الكائن، وعندما يتم تمرير الكائن مرة أخرى إلى الإجرائية المستدعية، يتم إنشاء نسخة مرة أخرى. من أجل الكائنات الكبيرة، هذا النسخ يمكن أن يستهلك الوقت والموارد الأخرى. وبالتالي، عند تمرير الكائنات إلى إجرائية، ترغب عادة بتجنب نسخ الكائن، مما يعني أنك ترغب بتمريرها بالمرجع بدلا من القيمة.

---

1 **إشارة مرجعية** للحصول على المزيد من النصائح حول استخدام المؤشرات في "سي++"، راجع "سي++" الفعالة، الإصدار الثاني (مايرز 1998) (effective C++، 2d ed. (Meyers 1998)، و"سي++" أكثر فعالية (مايرز 1996) (More Effective C++ (Meyers 1996).

ومع ذلك، في بعض الأحيان، ترغب بالحصول على الدلالات، التمرير بالقيمة – وهي أن الكائن الممرّر لا يجب أن يتم تغييره، مع تنفيذ التمرير بالمرجع – تمرير الكائن الأصلي بدلا من تمرير نسخة عنه.

في "سي++"، فإن تحليل هذه المسألة هو أنك تستخدم المؤشرات للتمرير بالمرجع – قد تبدو المصطلحات غريبة – و"المراجع الثابتة" للتمرير بالقيمة! إليك مثال على ذلك:

مثال في سي++ عن تمرير الوسطاء بالمرجع وبالقيمة

```
void SomeRoutine(
const LARGE_OBJECT &nonmodifiableObject,
LARGE_OBJECT *modifiableObject
);
```

يوفر هذا النهج فائدة إضافية من توفير التمايز النحوي ضمن الإجرائية المستدعية بين الكائنات التي من المفترض أن تُعامل على أنها قابلة للتعديل وتلك التي ليست كذلك. في كائن قابل للتعديل، المراجع إلى الأعضاء سوف تستخدم التدوين "object->member"، في حين أنه بالنسبة للكائنات غير القابلة للتعديل، المراجع الى الأعضاء سوف تستخدم التدوين "object.member".

ما يقيد هذا النهج هو صعوبة نشر مراجع ثابتة. إذا كنت تتحكم في قاعدة التعليمات البرمجية الخاصة بك، فمن الانضباط الجيد استخدام ثابت كلما كان ذلك ممكنا (مايرز 1998)، ويجب أن تكون قادرا على تصريح وسطاء تمرير بالقيمة كمراجع ثابتة. بالنسبة إلى شفرة المكتبة أو شفرة أخرى لا تتحكم فيها، ستواجه مشكلات في استخدام وسطاء الاجرائية الثابتة. الوضع الاحتياطي هو الإبقاء على استخدام المراجع لوسطاء القراءة فقط ولكن عدم التصريح عنهم كثوابت. مع هذا النهج، فإنك لن تدرك الفوائد الكاملة لتحقق المترجم من محاولات التعديل على وسطاء إجرائية غير قابلة للتعديل، ولكن على الأقل تعطي لنفسك التمييز البصري بين "object->member"و "object.member".

استخدم المؤشرات الآلية(*auto_ptrs*) إذا لم تكن قد طورت عادة استخدام المؤشرات الآلية، تعوّد عليها! بحذف الذاكرة تلقائيا عندما يخرج مؤشر آلي من النطاق، المؤشرات الآلية تجنب العديد من مشاكل تسرب-الذاكرة المرتبطة بالمؤشرات العادية. في سكوت مايرز سي++ أكثر فعالية، البند رقم 9 يحتوي على مناقشة جيدة للمؤشرات الآلية (مايرز1996).

كن ذكياً مع المؤشرات الذكية المؤشرات الذكية هي بديل للمؤشرات العادية أو المؤشرات "الغبية" (مايرز 1996). وهي تعمل على نحو مماثل للمؤشرات العادية، ولكنها توفر مزيداً من التحكم في إدارة الموارد وعمليات النسخ وعمليات الإسناد وبناء الكائنات وتدمير الكائنات. القضايا المعنية هي محددة لـ "سي++". "سي++" أكثر فعالية، البند رقم 28، يحتوي على مناقشة كاملة.

مؤشرات مؤشر-سي

فيما يلي بعض النصائح حول استخدام المؤشرات التي تنطبق تحديداً على لغة "سي":

استخدم أنواع مؤشرات محددة بدلا من النوع الافتراضي لغة "سي" تتيح لك استخدام مؤشرات char أو void لأي نوع من المتغيرات. وطالما أن المؤشر يؤشر، فإن اللغة لا تهتم حقاً بما يشير إليه. إذا كنت تستخدم أنواعاً محددة لمؤشراتك، مع ذلك، فالمترجم يمكن أن يعطيك تحذيرات حول عدم تتطابق أنواع المؤشرات والاختلافات غير اللائقة. إذا لم تستخدم الأنواع المحددة، فالمترجم لا يمكنه ذلك. استخدم نوع المؤشر المحدد متى أمكن ذلك.

النتيجة اللازمة لهذه القاعدة هي استخدام تحويل نوع واضح عندما يكون عليك القيام بعملية تحويل نوع. على سبيل المثال، في قطعة الشفرة هذه، من الواضح أنه يتم تحصيص متغير من النوع "NODE_PTR":

مثال في "سي" على قصر النوع المحدد

```
NodePtr = (NODE_PTR) calloc( 1, sizeof( NODE ) );
```

تجنب تحويل النوع (*type casting*) تجنب تحويل النوع لا علاقة له بالذهاب إلى مدرسة التمثيل أو الخروج من أن تلعب دوما "الثقيل". له علاقة بتجنب ضغط متغير من نوع معين في فضاء متغير من نوع آخر. تحويل النوع يوقف قدرة المترجم الخاص بك للتحقق من عدم تطابق الأنواع، وبالتالي يخلق ثغرة في درع البرمجة الوقائية الخاصة بك. البرنامج الذي يتطلب العديد من تحويلات الأنواع من المحتمل أن يكون فيه بعض الثغرات المعمارية التي تحتاج إلى إعادة النظر. أو إعادة التصميم إذا كان ذلك ممكنا. وإلا، حاول تجنب تحويل الأنواع بقدر المستطاع.

اتبع قاعدة "العلامة النجمية" لتمرير الوسطاء يمكنك تمرير وسيط عائد من إجرائية في "سي" فقط إذا كان لديك علامة نجمية (*) أمام الوسيط في عبارة التعيين. ويواجه العديد من مبرمجي "سي" صعوبة في تحديد متى تسمح "سي" بتمرير قيمة عائدة إلى الإجرائية المستدعية. من السهل تذكر أنه طالما لديك علامة النجمة أمام الوسيط عند تعيينه قيمة، يتم تمرير قيمة عائدة إلى الاجرائية المستدعية. بغض النظر عن عدد علامات النجمة التي تكدسها في التصريح، يجب أن يكون لديك واحدة على الأقل في عبارة التعيين إذا كنت ترغب في تمرير قيمة عائدة. على سبيل المثال، في قطعة الشفرة التالية، لا يتم تمرير القيمة المعينة للوسيط عائدة إلى الإجرائية المستدعية لأن عبارة التعيين لا تستخدم علامة النجمة:

مثال في "سي" على تمرير وسيط لن يعمل

```
void TryToPassBackAValue( int *parameter ) {
    parameter = SOME_VALUE;
}
```

هنا، يتم تمرير القيمة المعينة للوسيط مرة أخرى لأن الوسيط يحتوي على علامة نجمية أمامه:

مثال في "سي" على تمرير وسيط سوف يعمل

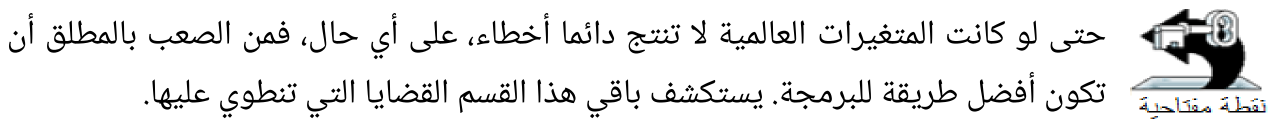
```
void TryToPassBackAValue( int *parameter ) {
    *parameter = SOME_VALUE;
}
```

استخدم "sizeof()" لتحديد حجم متغير في تخصيص الذاكرة إنه من الأسهل استخدام "sizeof()" من البحث عن الحجم في الدليل، و"sizeof()" تعمل للهياكل التي تنشئها بنفسك، والتي ليست في الدليل. لأنه يتم حسابها في وقت الترجمة، " sizeof()" لا تحمل عقوبة الأداء. انها محمولة - تعيد الترجمة في بيئة مختلفة تلقائياً وتغير القيمة التي يحسبها "sizeof()". ويتطلب صيانة قليلة حيث يمكنك تغيير الأنواع التي عرفتها وسيتم تعديل المخصصات تلقائياً.

13. 3 البيانات الشاملة(العامة)

المتغيرات الشاملة يمكن الوصول إليها في أي مكان في البرنامج.[1] كما يستخدم المصطلح أحياناً بإهمال للإشارة إلى المتغيرات ذات نطاق أوسع من المتغيرات المحلية – مثل متغيرات الصف التي يمكن الوصول إليها في أي مكان داخل الصف. لكن إمكانية الوصول في أي مكان داخل صف واحد لا تعني بحد ذاتها أن المتغير شامل.

أكثر المبرمجين الخبرة توصّلوا إلى أن استخدام البيانات الشاملة أكثر خطورة من استخدام البيانات المحلية. وقد توصّل أكثر المبرمجين خبرة أيضاً إلى أن الوصول إلى البيانات من عدة إجرائيات مفيد جدا.





حتى لو كانت المتغيرات العالمية لا تنتج دائما أخطاء، على أي حال، فمن الصعب بالمطلق أن تكون أفضل طريقة للبرمجة. يستكشف باقي هذا القسم القضايا التي تنطوي عليها.

المشاكل الشائعة مع البيانات الشاملة

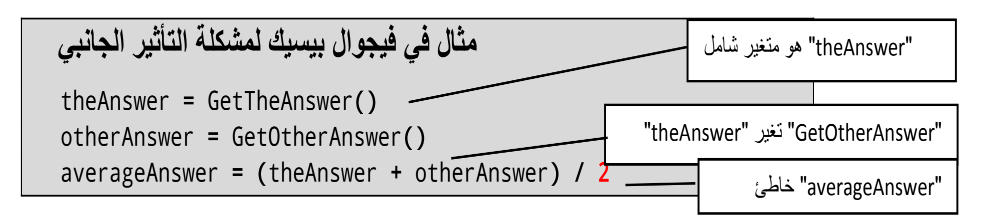
إذا كنت تستخدم المتغيرات الشاملة عشوائيا أو كنت تشعر بأنك مقيد بعدم القدرة على استخدامها، فلربما لم تدرك القيمة الكاملة لإخفاء المعلومات والنمطية حتى الآن. النمطية، وإخفاء المعلومات وما يرتبط بهما من استخدام الصفوف المصممة بشكل جيد قد لا يكشف الحقائق، ولكنها تقطع شوطا طويلا نحو جعل البرامج

[1] **إشارة مرجعية** للحصول على تفاصيل حول الفروق بين البيانات الشاملة وبيانات الصف، انظر "بيانات الصف الخاطئة للبيانات الشاملة" في القسم 5-3.

الكبيرة مفهومة وقابلة للصيانة. بمجرد فهمك للرسالة، سوف ترغب بكتابة إجرائيات وصفوف مع أقل قدر ممكن من الاتصالات مع المتغيرات الشاملة والعالم الخارجي.

يشير الناس إلى العديد من المشاكل في استخدام البيانات الشاملة، ولكن المشاكل تتضاءل إلى عدد صغير من القضايا الرئيسية:

التغييرات غير المقصودة على البيانات الشاملة قد تغير قيمة متغير شامل في مكان واحد وتعتقد خطأ أنها بقيت دون تغيير في مكان آخر. تعرف هذه المشكلة باسم "التأثير الجانبي". على سبيل المثال، في هذا المثال، "theAnswer" هو متغير شامل:



مثال في فيجوال بيسيك لمشكلة التأثير الجانبي

```
theAnswer = GetTheAnswer()
otherAnswer = GetOtherAnswer()
averageAnswer = (theAnswer + otherAnswer) / 2
```


قد تفترض أن استدعاء"GetOtherAnswer()" لا يغير قيمة "theAnswer"؛ إذا كان يغير، فإن المتوسط (المعدل) في السطر الثالث سيكون خطأ. وفي الواقع "GetOtherAnswer()" بالفعل تغير قيمة "theAnswer"، وبالتالي فإن البرنامج لديه خطأ يجب أن يتم إصلاحه.

مشاكل الأسماء الغريبة والأسماء المستعارة للبيانات الشاملة "الاسم المستعار" يشير إلى تسمية نفس المتغير باثنين أو أكثر من الأسماء المختلفة. يحدث هذا عندما يتم تمرير متغير شامل إلى إجرائية ثم يستخدم من قبل الإجرائية كمتغير شامل وكوسيط. إليك إجرائية تستخدم متغيرا شاملاً:

مثال في فيجوال بيسيك على إجرائية مؤاتية لمشكلة الاسم المستعار

```
Sub WriteGlobal( ByRef inputVar As Integer )
    inputVar = 0
    globalVar = inputVar + 5
    MsgBox( "Input Variable: " & Str( inputVar ) )
    MsgBox( "Global Variable: " & Str( globalVar ) )
End Sub
```

فيما يلي الشفرة التي تستدعي الإجرائية مع المتغير الشامل كوسيط:

مثال في فيجوال بيسيك على استدعاء الإجرائية مع وسيط، والتي تعرض مشكلة الاسم المستعار

```
WriteGlobal( globalVar )
```

منذ أنه تمت تهيئة "inputVar" بالـ"0" وأن " WriteGlobal()" تضيف "5" إلى "inputVar" للحصول على "globalVar"، سوف تتوقع أن يكون "globalVar" أكثر بـ"5" من "inputVar". ولكن هذه هي النتيجة المفاجئة:

نتيجة مشكلة الاسم المستعار في فيجوال بيسيك

```
Input Variable: 5
Global Variable: 5
```

الدقة هنا هو أن "globalVar" و"inputVar" هي في الواقع نفس المتغيرا! منذ أنه قد تم تمرير "globalVar" إلى "WriteGlobal()" من قبل الإجرائية المستدعية، فإنه قد تمت الإشارة إليه أو "استعارته" باسمين مختلفين. وبالتالي فإن تأثير أسطر صندوق الرسالة "MsgBox()" يختلف تماما عن الأسطر المقصودة: فهي تعرض نفس المتغير مرتين، على الرغم من أنها تشير إلى اسمين مختلفين.



إعادة إدخال مشاكل الشفرة مع البيانات الشاملة الشفرة التي يمكن إدخالها من قبل أكثر من مجرى "خط" (Thread) تحكم واحد أصبحت شائعة على نحو متزايد. وتخلق الشفرة متعددة المجاري (Multithreaded) إمكانية مشاركة البيانات الشاملة ليس فقط بين الإجرائيات، ولكن بين نسخ مختلفة من نفس البرنامج. في مثل هذه البيئة، عليك التأكد من أن البيانات الشاملة تبقي مضمونها حتى عندما يتم تشغيل عدة نسخ من البرنامج. هذه مشكلة كبيرة، ويمكنك تجنبها باستخدام التقنيات المقترحة لاحقاً في هذا القسم.

أعاقت البيانات الشاملة إعادة استخدام الشفرة لاستخدام الشفرة من أحد البرامج في برنامج آخر، عليك أن تكون قادر على سحبها من البرنامج الأول ووصلها في الثاني. من الناحية المثالية، سوف تكون قادر على رفع إجرائية واحدة أو صف واحد، توصيله في برنامج آخر، وتكمل طريقك بمرح.

البيانات الشاملة عقّدت الصورة. إذا كان الصف الذي تريد إعادة استخدامه يقرأ أو يكتب بيانات شاملة، لا يمكنك ببساطة وصله في البرنامج الجديد. عليك تعديل البرنامج الجديد أو الصف القديم بحيث يصبحان متلائمان. إذا سلكت الطريق الصعب، ستقوم بتعديل الصف القديم بحيث لا يستخدم البيانات الشاملة. إذا قمت بذلك، في المرة القادمة التي تحتاج إلى إعادة استخدام الصف سوف تكون قادراً على وصله بدون أي ضجة إضافية. وإذا سلكت الطريق السهل، فسوف تقوم بتعديل البرنامج الجديد لإنشاء البيانات الشاملة التي يحتاج الصف القديم لاستخدامها. هذا مثل الفيروس. فالبيانات الشاملة لا تؤثر على البرنامج الأصلي فقط، ولكن ينتشر التأثير أيضاً إلى البرامج الجديدة التي تستخدم أي من صفوف البرنامج القديم.

قضايا تراتيب التهيئة غير المؤكدة للبيانات الشاملة لا يتم تعريف الترتيب الذي يتم فيه تهيئة البيانات بين (ملفات) "وحدات الترجمة" المختلفة في بعض اللغات، لا سيما "سي++". إذا كانت تهيئة متغير شامل في أحد الملفات تستخدم متغير شامل تتم تهيئته في ملف مختلف، كافة الرهانات متوقفة على قيمة المتغير الثاني ما لم تتخذ خطوات محددة لضمان تهيئة المتغيرين في التسلسل الصحيح.

هذه المشكلة قابلة للحل مع الحل الذي يصفه سكوت مايرز في "سي++" فعّالة، البند رقم 47 (مايرز 1998). ولكن خدعة الحل هو ممثل للتعقيد الإضافي الذي يطرحه استخدام البيانات الشاملة.

النمطية والإدارة الفكرية تضررت من البيانات الشاملة جوهر إنشاء البرامج التي هي أكبر من بضع مئات من أسطر التعليمات البرمجية هو إدارة التعقيد. الطريقة الوحيدة التي يمكنك إدارة برنامج كبير فكرياً هو كسره إلى قطع بحيث عليك التفكير في جزء واحد فقط في وقت واحد. البنيويّة هي أقوى أداة تحت تصرفك لكسر البرنامج إلى قطع.

البيانات الشاملة تحدث فجوات في قدرتك على النمطية. إذا كنت تستخدم البيانات الشاملة، هل يمكنك التركيز على إجرائية واحدة في وقت واحد؟ لا. يجب عليك التركيز على إجرائية واحدة وكل إجرائية اخرى تستخدم نفس البيانات الشاملة. على الرغم من أن البيانات الشاملة لا تدمر تماماً نمطية البرنامج، إلّا أنها تضعفها، وهذا سبب كافي لكي تحاول إيجاد حلول أفضل لمشاكلك.

أسباب استخدام البيانات الشاملة

يجادل أصحاب البيانات أحيانا بأن المبرمجين يجب ألا يستخدموا البيانات الشاملة، ولكن معظم البرامج تستخدم "البيانات الشاملة" عندما يفسَّر المصطلح على نطاق واسع. البيانات في قاعدة البيانات هي بيانات شاملة، كما هو حال البيانات في ملفات الإعدادات (التكوين) مثل سجل ويندوز. الثوابت المسماة هي بيانات شاملة، ولكنها ليست متغيرات شاملة.

فباستخدامها بانضباط ستكون المتغيرات الشاملة مفيدة في العديد من الحالات:

الحفاظ على القيم الشاملة في بعض الأحيان لديك بيانات تنطبق من الناحية المفاهيمية على البرنامج بأكمله. قد يكون هذا متغيرا يعكس حالة البرنامج – على سبيل المثال، وضع تفاعلي مقابل سطر الأوامر، أو وضع طبيعي مقابل وضع استرداد الخطأ. أو قد تكون معلومات مطلوبة على طول البرنامج – على سبيل المثال، جدول بيانات تستخدمه كل إجرائية في البرنامج.

محاكاة الثوابت المسماة[1] على الرغم من أن "سي++" وجافا وفيجوال بيسيك، ومعظم اللغات الحديثة تدعم الثوابت المسماة، إلاّ أن بعض اللغات مثل "بايثون" و"بيرل" و"أوك" و"يونيكس شيل سكريبت" مازالت لا تدعما. يمكنك استخدام المتغيرات الشاملة كبديل للثوابت المسماة عندما لا تدعمها اللغة الخاصة بك. على سبيل المثال، يمكنك استبدال القيم الحرفية "1" و"0" بتعيين المتغيرين الشاملين "TRUE" و"FALSE" إلى "1"

[1] **إشارة مرجعية** لمزيد من التفاصيل حول الثوابت المسماة، راجع القسم 12-7 "الثوابت المسماة".

و"0"، أو يمكنك استبدال "66" كعدد الأسطر في الصفحة بـ "LINES_PER_PAGE = 66". من الأسهل تغيير الشفرة لاحقاً عند استخدام هذا النهج، وتميل الشفرة إلى أن تكون أسهل للقراءة. هذا الاستخدام المنضبط للبيانات الشاملة هو مثال رئيسي للتمييز بين البرمجة في لغة مقابل البرمجة بلغة، والتي يتم مناقشتها أكثر في القسم 34-4، "برمج في لغتك، وليس بها".

محاكاة أنواع التعداد يمكنك أيضا استخدام المتغيرات الشاملة لمحاكاة أنواع التعداد بلغات مثل "بايثون" التي لا تدعم أنواع التعداد مباشرة.

تنظيم استخدام البيانات الشائعة للغاية في بعض الأحيان يكون لديك العديد من المرجعيات إلى متغير يظهر في قائمة الوسطاء لكل إجرائية تكتبها. فبدلا من تضمينه في كل قائمة وسطاء، يمكنك جعله متغيراً شاملاً. ومع ذلك، في الحالات التي يبدو فيها أن متغير يمكن الوصول إليه في كل مكان، ونادرا ما يكون. فعادة ما يتم الوصول إليه من قبل مجموعة محدودة من الإجرائيات، يمكنك حزمه في صف مع البيانات التي تعمل عليها. المزيد عن هذا لاحقا.

إزالة البيانات المتشردة في بعض الأحيان تقوم بتمرير البيانات إلى إجرائية أو صف فقط لكي يمكن تمريرها إلى إجرائية اخرى أو صف. على سبيل المثال، قد يكون لديك كائن معالجة خطأ يتم استخدامه في كل إجرائية. عندما لا تستخدم الإجرائية الكائن في وسط سلسلة الاستدعاءات، يطلق على الكائن "بيانات متشردة". استخدام المتغيرات الشاملة يمكن أن يقضي على البيانات المتشردة.

استخدم البيانات الشاملة فقط كملجأ أخير

قبل أن تلجأ إلى استخدام البيانات الشاملة، ضع في اعتبارك بعض البدائل:

ابدأ بجعل كل متغير محلياً، واجعل المتغيرات شاملة فقط كلما تحتاج إليها اجعل جميع المتغيرات محلية إلى الإجرائيات بشكل فردي في البداية. وإذا وجدت أن هناك حاجة إليها في مكان آخر، اجعلها متغيرات صف خاصة أو محمية قبل أن تذهب إلى حد جعلها شاملة. إذا وجدت أخيرا أنه يجب عليك جعلها شاملة، افعل ذلك، ولكن فقط عندما تكون متأكداً أنه يجب عليك ذلك. إذا بدأت بجعل متغير شاملاً، سوف لن تجعله محلياُ أبداً، في حين إذا بدأت بجعله محلياً، قد لا تحتاج أبداً لجعله شاملاً.

ميِّز بين المتغيرات الشاملة ومتغيرات الصف بعض المتغيرات شاملة حقا لأنه يتم الوصول إليها خلال كامل البرنامج. البعض الآخر هي في الواقع متغيرات صف، تستخدم بشكل كبير فقط ضمن مجموعة معينة من

الإجرائيات. لا بأس بالوصول إلى متغير صف بالطريقة التي تريدها ضمن مجموعة من الإجرائيات التي تستخدمه بشكل كبير. إذا كانت الإجرائيات خارج الصف تحتاج إلى استخدامه، قم بتوفير قيمة المتغير عن طريق إجرائية وصول. لا تقم بالوصول إلى قيم الصف بشكل مباشر – كما لو كانت متغيرات عالمية – حتى لو كانت لغة البرمجة تسمح لك بذلك. هذه النصيحة هي بمثابة القول "وحّد! وحّد! وحّد".

استخدم إجرائيات الوصول إنشاء إجرائيات الوصول هو "حصان الشغل" (أساس) للتغلب على المشاكل مع البيانات الشاملة. المزيد عن ذلك في القسم التالي.

استخدام إجرائيات الوصول بدلا من البيانات الشاملة



إن أي شيء يمكنك القيام به مع البيانات الشاملة، يمكنك أن تقوم به بشكل أفضل مع إجرائيات الوصول. استخدام إجرائيات الوصول هو تقنية أساسية لتنفيذ أنواع البيانات المجردة وتحقيق إخفاء المعلومات. حتى لو لم تكن ترغب باستخدام نوع بيانات مجرد كاملاً، لا يزال بإمكانك استخدام إجرائيات الوصول لمركزة السيطرة على البيانات الخاصة بك وحماية نفسك من التغييرات.

ميزات إجرائيات الوصول

استخدام إجرائيات الوصول له ميزات متعددة:

- تحصل على السيطرة المركزية على البيانات. إذا اكتشفت تنفيذ أكثر ملاءمة للهيكل في وقت لاحق، لا يجب عليك تغيير الشفرة في كل مكان تتم فيه الإشارة إلى البيانات. التغييرات لا تتموج خلال البرنامج بأكمله. بل يبقون داخل إجرائيات الوصول.
- يمكنك التأكد من أن جميع المرجعيات إلى المتغير مقيّدة[1]. إذا كنت تدفع العناصر في المكدس بواسطة عبارات مثل "stack.array[stack.top]=newElement"، يمكنك أن تنسى بسهولة التحقق من طفحان المكدس وتقوم بخطأ خطير. إذا كنت تستخدم إجرائيات الوصول – على سبيل المثال، "PushStack(newElement)" – يمكنك كتابة التحقق من طفحان المكدس في إجرائية "PushStack()". و سوف يتم التحقق تلقائياً في كل مرة يتم فيها استدعاء الإجرائية، ويمكنك نسيان ذلك.
- يمكنك الحصول على الفوائد العامة من إخفاء المعلومات تلقائياً.[2] إجرائيات الوصول هي مثال على إخفاء المعلومات، حتى لو لم تقم بتصميمها لهذا السبب. يمكنك تغيير المناطق الداخلية من إجرائية

[1] **إشارة مرجعية** لمزيد من التفاصيل حول المتاريس، انظر القسم 8-5، "حصّن برنامجك لاحتواء الضرر الناجم عن الأخطاء".

[2] **إشارة مرجعية** لمزيد من التفاصيل حول إخفاء المعلومات، انظر "إخفاء الأسرار (إخفاء المعلومات)" في القسم 5-3.

الوصـول دون تغيير بقية البرنامج. إجرائيات الوصـول تسـمح لك بإعادة تزيين المناطق الداخلية من منزلك وترك الخارج دون تغيير بحيث لا يزال بإمكان أصدقائك التعرف عليه.

- من السـهل تحويل إجرائيات الوصـول إلى نوع بيانات مجرد. أحد ميزات إجرائيات الوصـول هو أنه يمكنك إنشاء مستوى تجريد يصعب القيام به عندما تعمل مع البيانات الشاملة مباشرة. على سبيل المثال، بدلا من كتابة الشفرة التي تقول "if lineCount>MAX_COUNT"، إجرائية الوصـول تسـمح لك بكتابة الشفرة التي تقول "*if PageFull* ()". هذا التغيير الصغير يوثق القصد من اختبار "PageFull()"، وهو يفعل ذلك في الشفرة. انها مكسب صغير في سهولة القراءة، ولكن الاهتمام المستمر بمثل هذه التفاصيل يصنع الفرق بين البرمجيات المصنوعة بشكل جميل والشفرة المتلائمة مع بعضها فقط.

كيفية استخدام إجرائيات الوصول

وإليك النسخة القصيرة من النظرية والممارسة لإجرائيات الوصول: إخفاء البيانات في صف. عرّف تلك البيانات باستخدام الكلمة المفتاحية "static" أو ما يعادلها لضمان وجود حالة واحدة فقط من البيانات. اكتب الإجرائيات التي تمكنك من رؤية البيانات وتغييرها. اطلب من الشفرة خارج الصف استخدام إجرائيات الوصول بدلا من العمل مع البيانات بشكل مباشر.

على سبيل المثال، إذا كان لديك متغير الحالة العامة "g_globalStatus" الذي يصف الحالة العامة لبرنامجك، يمكنك إنشاء اثنين من إجرائيات الوصول: "globalStatus.Get()" و"globalStatus.Set()"، كل منها يفعل ما يبدو عليه أنه يفعل. هذه الإجرائيات تصل إلى متغير مخفي داخل الصف والذي يحل محل "g_globalStatus". بقية البرنامج يمكنه الحصول على كل الفائدة من المتغير الشامل السابق عن طريق الوصول إلى "globalStatus.Get()" و"globalStatus.Set()".

إذا كانت لغتك لا تدعم الصفوف[1]، لا يزال بإمكانك إنشاء إجرائيات الوصول للتعامل مع البيانات الشاملة ولكن سيكون عليك فرض قيود على استخدام البيانات الشاملة من خلال معايير الترميز بدلا من التقييد المدمج في لغة البرمجة.

فيما يلي بعض الإرشادات التفصيلية لاستخدام إجرائيات الوصول لإخفاء المتغيرات الشاملة عندما لا تحتوي لغتك على دعم مضمن:

[1] **إشارة مرجعية** تقييد الوصول إلى المتغيرات الشاملة حتى عندما تكون لغتك لا تدعم ذلك بشكل مباشر هو مثال عن البرمجة إلى لغة مقابل البرمجة في لغة. لمزيد من التفاصيل، راجع القسم 34-4، "برمج في لغتك، وليس بها".

الطلب من كل الشفرة بالمرور خلال إجرائيات الوصول من أجل البيانات هناك اتفاقية جيدة تتطلب أن تبدأ جميع البيانات الشاملة بالبادئة "g_"، وأن تطلب كذلك عدم وصول أي شفرة إلى متغير له البادئة "g_" باستثناء إجرائيات الوصول إلى المتغير. كل الشفرة الأخرى تصل إلى البيانات من خلال إجرائيات الوصول.

فقط لا تضع كل بياناتك الشاملة في نفس البرميل إذا وضعت كل البيانات الشاملة في كومة كبيرة وكتبت إجرائيات وصول لها، عندها تتخلص من مشاكل البيانات الشاملة ولكنك تفوت بعض من مزايا إخفاء المعلومات وأنواع البيانات المجردة. وطالما أنك تكتب إجرائيات الوصول، فاستغرق بعض الوقت للتفكير إلى أي صف ينتمي كل متغير شامل ثم قم بتجميع البيانات وإجرائيات الوصول الخاصة بها مع البيانات والإجرائيات الأخرى في ذلك الصف.

استخدم القفل (الإقفال) للتحكم في الوصول إلى المتغيرات الشاملة على غرار التحكم في التزامن في بيئة قاعدة بيانات متعددة المستخدمين، يتطلب القفل (الإقفال) أنه قبل أن يتم استخدام قيمة متغير شامل أو تحديثه، يجب أن يكون المتغير قد تم "سحبه". بعد استخدام المتغير، يتم إدخاله مرة أخرى. خلال الوقت الذي يتم فيه استخدام المتغير (سحب)، إذا حاول جزء آخر من البرنامج سحبه، إجرائية "القفل/فتح القفل" تعرض رسالة خطأ أو تطلق تأكيداً.

هذا الوصف للإقفال يتجاهل العديد من خفايا كتابة الشفرة لدعم التزامن بشكل كامل.[1] ولهذا السبب، فإن مخططات الإقفال المبسطة من هذا القبيل هي الأكثر فائدة خلال مرحلة التطوير. ما لم يكن مخططاً جيداً جداً، فإنه على الأرجح لن يكون موثوقاً بما يكفي لوضعه في الإنتاج. عندما يتم وضع البرنامج في الإنتاج، يتم تعديل الشفرة للقيام بشيء أكثر أمانا وأكثر رشاقة من عرض رسائل الخطأ. على سبيل المثال، قد تسجل رسالة خطأ إلى ملف عندما تكتشف أن هناك أجزاء متعددة من البرنامج تحاول قفل نفس المتغير الشامل.

هذا النوع من الحماية في وقت التطوير من السهل إلى حد ما تنفيذه عندما تستخدم إجرائيات الوصول للبيانات الشاملة، ولكن سيكون من غير الملائم لتنفيذه إذا كنت تستخدم البيانات الشاملة مباشرة.

قم ببناء مستوى من التجريد في إجرائيات الوصول الخاصة بك قم ببناء إجرائيات الوصول على مستوى مجال المشكلة بدلاً من مستوى تفاصيل التنفيذ. هذا النهج يشتري لك سهولة قراءة محسّنة وكذلك التأمين ضد التغييرات في تفاصيل التنفيذ.

قارن بين أزواج العبارات في الجدول 13-1:

1 **إشارة مرجعية** لمزيد من التفاصيل حول الفروق بين نسخ التطوير والإنتاج للبرنامج، انظر "خطة لإزالة أدوات التصحيح" في القسم 8-6 والقسم 8-7، "تحديد كمية البرمجة الوقائية المتروكة في الشفرة النهائية".

الجدول 13-1 الوصول إلى البيانات الشاملة مباشرة ومن خلال إجرائيات الوصول

الاستخدام المباشر للبيانات الشاملة	استخدام البيانات الشاملة من خلال إجرائيات الوصول
node = node.next	account = NextAccount(account)
node = node.next	employee = NextEmployee(employee)
node = node.next	rateLevel = NextRateLevel(rateLevel)
event = eventQueue[queueFront]	event = HighestPriorityEvent()
event = eventQueue[queueBack]	event = LowestPriorityEvent()

في الأمثلة الثلاثة الأولى، النقطة هي أن إجرائية وصول مجردة تخبرك أكثر بكثير من بنية عامة. إذا استخدمت البنية مباشرة، فإنك تفعل الكثير في آن واحد: أنت تظهر معاً – ما يفعله الهيكل نفسه (الانتقال إلى العقدة التالية في لائحة مترابطة) – وما يتم عمله فيما يتعلق بالكيان الذي يمثله (الحصول على حساب، الموظف التالي، أو مستوى التقييم). وهذا عبء كبير لوضعه على مهمة بسيطة في هيكل البيانات. إخفاء المعلومات وراء إجرائيات وصول مجردة يتيح للشفرة أن تتحدث عن نفسها ويجعل الشفرة تقرأ على مستوى مجال المشكلة، وليس على مستوى تفاصيل التنفيذ.

إبقاء جميع سماحيات الوصول إلى البيانات في نفس مستوى التجريد إذا استخدمت إجرائية وصول للقيام بشيء واحد على بنية، يجب عليك استخدام إجرائية وصول للقيام بكل شيء آخر عليها أيضاً. إذا قرأت من البنية باستخدام إجرائية وصول، اكتب عليها باستخدام إجرائية وصول. إذا قمت باستدعاء "`InitStack()`" لتهيئة مكدس و"`PushStack()`" لدفع عنصر إلى المكدس، فقد قمت بإنشاء عرض متناسق للبيانات. إذا قمت بالسحب من المكدس من خلال كتابة "`value=array[stack.top]`"، فقد قمت بإنشاء عرض غير متناسق للبيانات. إن التناقض يجعل من الصعب على الآخرين فهم الشفرة. أنشئ إجرائية "`PopStack()`" بدلا من كتابة "`value=array[stack.top]`".

في المثال أزواج العبارات في الجدول 13-1، حدثت عمليتي رتل-الأحداث على التوازي.[1] إدراج حدث في الرتل سيكون أكثر صعوبة من أي من العمليتين في الجدول، مما يتطلب عدة أسطر من التعليمات البرمجية للعثور على المكان لإدراج الحدث، وضبط الأحداث الموجودة لإفساح مجال للحدث الجديد، وضبط مقدمة أو مؤخرة الرتل. إزالة حدث من الرتل سيكون معقداً مثله تماماً. خلال كتابة الشفرة، سيتم وضع العمليات المعقدة

1 **إشارة مرجعية** استخدام إجرائيات الوصول لرتل أحداث يقترح الحاجة لإنشاء صف. لمزيد من التفاصيل، راجع الفصل 6 "الصفوف الناجحة".

في إجرائيات، والأخرى سوف تترك كتعديل مباشر على البيانات. وهذا سيخلق استخدام قبيح وغير متوازٍ للهيكل. الآن قارن بين أزواج العبارات في الجدول 13-2:

الجدول 13-2 الاستخدامات المتوازية وغير المتوازية للبيانات المعقدة

استخدام غير متوازي للبيانات المعقدة	استخدام متوازي للبيانات المعقدة
event = EventQueue[queueFront]	event = HighestPriorityEvent()
event = EventQueue[queueBack]	event = LowestPriorityEvent()
AddEvent(event)	AddEvent(event)
eventCount = eventCount - 1	RemoveEvent(event)

على الرغم من أنك قد تعتقد أن هذه المبادئ التوجيهية تنطبق فقط على البرامج الكبيرة، فقد أظهرت إجرائيات الوصول نفسها لتكون وسيلة منتجة لتجنب مشاكل البيانات الشاملة. وكقيمة إضافية، إنها تجعل الشفرة أكثر قابلية للقراءة وتضيف المرونة.

كيف تقلل مخاطر استخدام البيانات الشاملة

في معظم الحالات، البيانات الشاملة هي حقا بيانات صف لصف لم يتم تصميمه أو تنفيذه بشكل جيد. في حالات قليلة، البيانات بحاجة فعلاً أن تكون شاملة، ولكن الوصول إليها يمكن أن يُغلف بإجرائيات الوصول لتقليل المشاكل المحتملة. في عدد قليل من الحالات المتبقية، أنت بحاجة حقّاً لاستخدام البيانات الشاملة. في هذه الحالات، قد تفكر في اتباع المبادئ التوجيهية في هذا القسم مثل شرب الماء على جرعات عند السفر إلى بلد أجنبي: انها عملية شاقة نوعاً ما، ولكنها تحسن احتمالات البقاء في صحة جيدة.

طوّر اصطلاحية تسمية تجعل المتغيرات الشاملة واضحة[1] يمكنك تجنب بعض الأخطاء فقط من خلال جعله من الواضح أنك تعمل مع البيانات الشاملة. إذا كنت تستخدم المتغيرات الشاملة لأكثر من غرض واحد (على سبيل المثال، كمتغيرات وكبدائل للثوابت المسماة)، فتأكد من أن اصطلاحية التسمية تفرّق بين أنواع الاستخدامات.

أنشئ قائمة مشروحة جيدا لجميع متغيراتك الشاملة بمجرد أن يشير اصطلاح التسمية الخاص بك إلى أن المتغير هو شامل، فمن المفيد الإشارة إلى ما يفعله المتغير. قائمة المتغيرات الشاملة هي واحدة من الأدوات الأكثر فائدة يمكن أن يحصل عليها شخص يعمل مع برنامجك.

1 **إشارة مرجعية** للحصول على تفاصيل حول اصطلاحات التسمية للمتغيرات الشاملة، راجع "تحديد المتغيرات الشاملة" في القسم 11-4.

لا تستخدم المتغيرات الشاملة لاحتواء نتائج وسيطة إذا كنت بحاجة إلى حساب قيمة جديدة لمتغير شامل، قم بتعيين المتغير الشامل للقيمة النهائية في نهاية الحساب بدلا من استخدامه للاحتفاظ بنتائج الحسابات الوسيطة.

لا تدعي أنك لا تستخدم البيانات الشاملة من خلال وضع كل بياناتك في كائن كبير وتمريره في كل مكان وضع كل شيء في كائن ضخم واحد قد يرضي المعنى الحرفي للقانون بتجنب المتغيرات الشاملة، لكنها نفقة زائدة صرفة، لا تنتج أي من فوائد التغليف الحقيقي. إذا كنت تستخدم البيانات الشاملة، فقم بذلك بشكل علني. لا تحاول أن تموه ذلك بالكائنات الكبيرة.

مصادر إضافية

فيما يلي المزيد من المصادر التي تغطي أنواع البيانات غير الاعتيادية:[1]

ماغواير، ستيف. كتابة شفرة صلبة. ريدموند، وا: مايكروسوفت بريس، 1993. يحتوي الفصل 3 على مناقشة ممتازة لمخاطر استخدام المؤشر والعديد من النصائح المحددة لتجنب المشاكل مع المؤشرات.

Maguire، Steve. Writing Solid Code. Redmond، WA: Microsoft Press، 1993 .

مايرز، سكوت. سي++ فعّالة الاصدار الثاني. قراءة، ما: أديسون-ويسلي، 1998؛ مايرز، سكوت، سي++ أكثر فعالية. قراءة، ما: أديسون-ويسلي، 1996. كما تشير العناوين، تحتوي هذه الكتب على العديد من النصائح المحددة لتحسين برامج سي++، بما في ذلك المبادئ التوجيهية لاستخدام المؤشرات بأمان وفعالية. سي++ أكثر فعالية على وجه الخصوص يحتوي على مناقشة ممتازة من قضايا إدارة الذاكرة في سي++.

Meyers، Scott. Effective C++، 2d ed. Reading، MA: Addison-Wesley، 1998; Meyers، Scott، More Effective C++. Reading، MA: Addison-Wesley، 1996 .

[1] cc2e.com/1385

قائمة المراجعة: اعتبارات في استخدام أنواع البيانات غير الاعتيادية [1]

الهياكل

- هل استخدمت هياكل بدلا من المتغيرات العارية لتنظيم ومعالجة مجموعات من البيانات المرتبطة؟
- هل فكرت في إنشاء صف كبديل لاستخدام هيكل؟

البيانات الشاملة

- هل جميع المتغيرات محلية أو في نطاق الصف ما لم تحتاج تماما إلى أن تكون شاملة؟
- هل تفرّق اصطلاحات تسمية المتغيرات بين البيانات المحلية، وبيانات الصف، والبيانات الشاملة؟
- هل تم توثيق جميع المتغيرات الشاملة؟
- هل الشفرة خالية من البيانات الشاملة الزائفة، والكائنات الضخمة التي تحتوي على خليط من البيانات التي تم تمريرها إلى كل إجرائية؟
- هل يتم استخدام إجرائيات الوصول بدلا من البيانات الشاملة؟
- هل يتم تنظيم إجرائيات الوصول والبيانات في صفوف؟
- هل توفر إجرائيات الوصول مستوى من التجريد يتجاوز التحقيقات الأساسية لأنماط البيانات؟
- هل جميع إجرائيات الوصول المرتبطة على نفس المستوى من التجريد؟

المؤشرات

- هل عمليات المؤشر معزولة في الإجرائيات؟
- هل مراجع المؤشر صالحة، أم يمكن أن يكون المؤشر معلّق؟
- هل تفحص الشفرة المؤشرات للتحقق من صحتها قبل استخدامها؟
- هل المتغير الذي يرجع إليه المؤشر يتم التحقق من صحته قبل استخدامه؟
- هل يتم تعيين المؤشرات إلى فارغة بعد تحريرها؟
- هل تستخدم الشفرة جميع متغيرات المؤشرات اللازمة من أجل سهولة القراءة؟
- هل يتم تحرير المؤشرات في اللوائح المترابطة بالترتيب الصحيح؟
- هل يقوم البرنامج بتخصيص مساحة احتياطية للذاكرة بحيث يمكن إيقافه بأمان إذا نفذت الذاكرة؟
- هل تستخدم المؤشرات فقط كملجأ أخير، عندما لا تتوفر طريقة أخرى؟

---

1 cc2e.com/1392

نقاط مفتاحية

- يمكن للهياكل أن تساعد على جعل البرامج أقل تعقيداً وأسهل للفهم وأسهل للصيانة.
- كلما فكرت في استخدام هيكل، خذ بالاعتبار ما إذا كان الصف سيعمل بشكل أفضل.
- المؤشرات عرضة للخطأ. أحمِ نفسك باستخدام إجرائيات الوصول أو الصفوف وممارسات البرمجة الوقائية.
- تجنب المتغيرات الشاملة، ليس فقط لأنها خطرة، ولكن لأنه يمكنك استبدالها بشيء أفضل.
- إذا لم تتمكن من تجنب المتغيرات الشاملة، أعمل معهم من خلال إجرائيات الوصول. إجرائيات الوصول تعطيك كل ما تعطيك إياه المتغيرات الشاملة، وأكثر.

القسم الثالث: العبارات

في هذا القسم:

الفصل الرابع عشر: تنظيم الشفرة الخطيّة

الفصل الخامس عشر: استخدام الشرطيّات

الفصل السادس عشر: التحكم بالحلقات

الفصل السابع عشر: بنى التحكم غير العادية

الفصل الثامن عشر: الطرق جدولية القيادة

الفصل التاسع عشر: قضايا التحكم العامة

الفصل 14

تنظيم الشفرة الخطيّة

المحتويات[1]

مواضيع ذات صلة

- مواضيع التحكم العامة: الفصل 19
- استخدام الشرطيّات: الفصل 15
- استخدام الحلقات: الفصل 16
- نطاق المتغيرات والكائنات: القسم 4.10،"النطاق"

ينتقل هذا الفصل من النظرة المركّزة على البيانات للبرمجة، إلى وجهة النظر المركزة على العبارات البرمجية. يُقدم هذا الفصل النوع الأبسط من التحكم بالتدفق: وضع العبارات البرمجية وكتل العبارات البرمجية في ترتيب متسلسل.

على الرغم من أن تنظيم الشفرة الخطيّة هو مهمة بسيطة نسبياً، ولكنه تؤثر بعض الخلافات التنظيمية على جودة الشفرة البرمجية، وعلى الاصلاح، وعلى قابلية القراءة، وعلى قابلية الصيانة.

1.14 العبارات البرمجية التي يجب أن تكون بترتيب محدد

إن أسهل العبارات البرمجية التسلسلية التي يمكن ترتيبها هي تلك التي يُحسب فيها الترتيب. فيما يلي مثال:

مثال بلغة البرمجة جافا عن العبارات البرمجية التي يُحسب قيها الترتيب

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

[1] cc2e.com/1465

إذا لم يحدث شيء ما غامض مع هذا المقطع من الشفرة البرمجية، فإنه يجب على العبارات البرميجة ان تُنفذ بالترتيب الظاهر. من المتوجب أن يتم قراءة البيانات قبل أن يتم حساب النتائج، ومن المتوجب أن يتم حساب النتائج قبل أن يتم طباعتها.

إن المفهوم الأساسي من هذا المثال هو بالتبعية. تعتمد العبارة الثالثة على العبارة الثانية، والثانية على الاولى. في هذا المثال، حقيقة اعتماد عبارة برمجية واحدة على أخرى هو واضح من أسماء الإجرائية. في المقطع البرمجي التالي، التبعية أقل وضوحاً:

مثال بلغة البرمجة جافا عن العبارات البرمجية التي يُحسب فيها الترتيب ولكن بشكل غير واضح

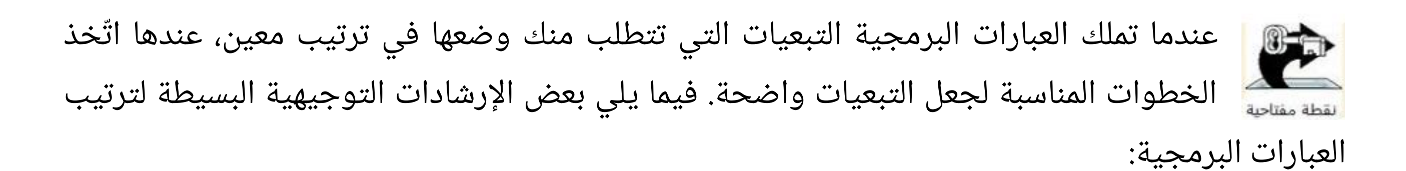
```
revenue.ComputeMonthly();
revenue.ComputeQuarterly();
revenue.ComputeAnnual();
```

في هذه الحالة، يفترض حساب الإيرادات الفصلية أنه قد تم للتو حساب الإرادات الشهرية. يُخبرك الإلمام بالمحاسبة - أو حتى الحس السليم – بأنه يجب حساب الإيرادات الفصلية قبل حساب الإيرادات السنوية. يوجد تبعية، ولكن لا يمكن ملاحظتها فقط بقراءة الشفرة. وهنا، التبعيات غير مُلاحظة – أنها مخفية بالمعنى الحرفي:

مثال بلغة البرمجة فيجول بيسك عن العبارات البرمجية التي فيها التبعيات مخفية

```
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

افترض أن الإجرائية ()ComputeMarketingExpense تُهيئ متغيرات عنصر الصف، التي توضع فيها بيانات كل الإجرائيات الأخرى. في مثل هكذا حالة، يجب استدعاء هذه الإجرائية قبل الإجرائيات الأخرى. كيف تستطيع أن تعرف هذا من قراءة هذه الشفرة؟ بما أنه لا تملك استدعاءات الإجرائية أية وسطاء، فإنك تستطيع أن تحزر أنه لدى كل واحدة من هذه الإجرائيات وصول إلى بيانات الصف. ولكنك لا تستطيع أن تعرف هذا بشكل اكيد من خلال قراءة هذه الشفرة.



عندما تملك العبارات البرمجية التبعيات التي تتطلب منك وضعها في ترتيب معين، عندها اتّخذ الخطوات المناسبة لجعل التبعيات واضحة. فيما يلي بعض الإرشادات التوجيهية البسيطة لترتيب العبارات البرمجية:

نظّم الشفرة بطريقة تكون فيها التبعيات واضحة. في مثال الفيجول بيسك، لا يجب على الإجرائية ()ComputeMarketingExpense أن تُهيئ متغيرات عنصر الصف. تقترح أسماء الإجرائية، بأن الإجرائية

ComputeMarketingExpense() مُشابهة للإجرائيات ComputeSalesExpense()، وComputeTravelExpense()، وتتوقع الإجرائيات الأخرى أن هذه الإجرائية تعمل مع بيانات التسويق (marketing)، بدلاً من العمل مع بيانات المبيعات أو أية بيانات أخرى. إن وجود الإجرائية ComputeMarketingExpense() التي تُهيئ متغير عنصر الصف، هو ممارسة خاطئة يجب عليك ان تتجنبها. لماذا يجب أن تتم عملية التهيئة في تلك الإجرائية بدلاً من واحدة من الإجرائيتين الآخرتين؟ بدلاً من التفكير بسبب جيد، فإنه يجب عليك أن تكتب الإجرائية الأخرى InitializeExpenseData() لتهيئة متغير عنصر الصف. إن اسم الإجرائية هو إشارة جيدة إلى أنه يجب أن يتم استدعائها قبل إجرائيات النفقات الاخرى.

سمّي الإجرائيات بطريقة تكون فيها التبعيات واضحة. في مثال الفيجول بيسك، إن الإجرائية ComputeMarketingExpense() مُسماة بطريقة غير مُناسبة، لأنها تقوم بأكثر من حساب لنفقات التسويق، بل إنها تقوم أيضاً بتهيئة بيانات العنصر. إذا كنت ضد إنشاء إجرائية إضافية لتهيئة البيانات، على الأقل قُم بإعطاء الإجرائية ComputeMarketingExpense() اسم مُناسب يصف كل الوظائف التي تُنجزها. في هذه الحالة، سيكون الاسم ComputeMarketingExpenseAndInitializeMemberData() مُناسب. من الممكن أن تقول إنه اسم فظيع، لأنه طويل جداً، ولكن يصف هذا الاسم ما تقوم به الإجرائية وهذا ليس بفظيع. بل الإجرائية بحد ذاتها فظيعة!

استخدم وسطاء الإجرائية لجعل التبعيات واضحة.[1] مرة ثانية في مثال الفيجول بيسك، بما أنه لا توجد بيانات يتم تمريرها بين الإجرائيات، فإنك لا تعرف أية إجرائيات تستخدم نفس البيانات. من خلال إعادة كتابة الشفرة، بحيث يتم تمرير البيانات بين الإجرائيات، فإنه يمكنك إعداد دليل على أن ترتيب التنفيذ مُهم. من الممكن أن تبدو الشفرة الجديدة بالشكل التالي:

مثال بلغة البرمجة فيجول بيسك عن البيانات التي تقترح تبعية الترتيب

```
InitializeExpenseData( expenseData )
ComputeMarketingExpense( expenseData )
ComputeSalesExpense( expenseData )
ComputeTravelExpense( expenseData )
ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

لأنه تستخدم كل الإجرائيات expenseData، فإنه لديك تلميح بأنه من الممكن أنها تعمل على نفس البيانات، وبأن ترتيب العبارات البرمجية من الممكن أن يكون مهم.

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن استخدام الإجرائيات ووسطائها، انظر الفصل 5 "التصميم في عملية البيناء".

في هذا المثال الخاص، الطريقة الأفضل هي بتحويل الإجرائية إلى توابع تأخذ expenseData كدخل وتُعيد expenseData المُحدثة كخرج، مما يجعل الأمر أكثر وضوحا أن الشفرة تتضمن تبعية الترتيب.

مثال بلغة البرمجة فيجول بيسك عن البيانات واستدعاءات الإجرائية التي تقترح تبعية الترتيب

```
expenseData = InitializeExpenseData( expenseData )
expenseData = ComputeMarketingExpense( expenseData )
expenseData = ComputeSalesExpense( expenseData )
expenseData = ComputeTravelExpense( expenseData )
expenseData = ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

يمكن أيضاً أن تُشير البيانات إلى أن تنفيذ الترتيب ليس بمهم، كما في هذه الحالة:

```
مثال بلغة البرمجة فيجول بيسك عن البيانات التي لا تشير إلى تبعية الترتيب
ComputeMarketingExpense( marketingData )
ComputeSalesExpense( salesData )
ComputeTravelExpense( travelData )
ComputePersonnelExpense( personnelData )
DisplayExpenseSummary( marketingData ،salesData ،travelData ،
personnelData )
```

بما أنهُ لا تملك الإجرائيات في السطور الأربعة الاولى أيّة بيانات مشتركة، فإنه تُشير الشفرة إلى ان الترتيب الذي يتم فيه استدعائها ليس بمهم. لأن الإجرائية تستخدم في السطر الخامس بيانات من كل من الإجرائيات الأربعة الأولى، فإنك تستطيع أن تفترض بأنه تحتاج إلى أن يتم تنفيذها بعد الإجرائيات الأربعة الاولى.

نقطة مفتاحية

وثّق التبعيات غير الواضحة مع تعليقات. حاول أولاً أن تكتب شفرة بدون تبعيات الترتيب. حاول ثانياً أن تكتب الشفرة التي تجعل هذه التبعيات واضحة. إذا لازلت تشعر بالقلق حول عدم وضوح تبعية الترتيب بما فيه الكفاية، وثّق هذا. إن توثيق التبعيات غير الواضحة هو أحد جوانب توثيق افتراضات كتابة الشفرة، التي هي ضرورية لكتابة شفرة قابلة للتعديل وقابلة للصيانة. في مثال الفيجول بيسك، ستكون التعليقات مع هذه السطور مفيدة:

مثال بلغة البرمجة فيجول بيسك عن عبارات برمجية التي فيها تبعيات الترتيب مخفية ولكنها موضّحة بتعليقات

```
' Compute expense data. Each of the routines accesses the
' member data expenseData. DisplayExpenseSummary
' should be called last because it depends on data calculated
' by the other routines.
InitializeExpenseData
```

```
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

لا تستخدم هذه الشفرة تقنيات جعل التبعيات واضحة. من المفضل الاعتماد على مثل هكذا تقنيات بدلاً من الاعتماد على التعليقات، ولكن إذا كنت تحافظ على الشفرة المتحكم بها بإتقان، أو إذا كنت لا تستطيع تحسين الشفرة بحد ذاتها لبعض الأسباب، فاستخدم التوثيق للتعويض عن نقاط الضعف في الشفرة.

تحقق من التبعيات مع المصادقات أو مع شفرة معالجة الخطأ. إذا كانت الشفرة حرجة بما فيه الكفاية، فمن الممكن أن تستخدم متغيرات الحالة وشفرة معالجة الخطأ أو المصادقات لتوثيق التبعيات المتسلسلة الحرجة. على سبيل المثال، في باني الصف، من الممكن أن تُهيأ متغير عنصر صف `isExpenseDataInitialized` بالقيمة `false`. ومن ثم في الإجرائية `tializeExpenseData()`، يمكنك أن تعد `isExpenseDataInitialized` إلى القيمة `true`. كل تابع يعتمد على `expenseData` جاري تهيئته يستطيع بعدها أن يتحقق فيما إذا كان `isExpenseDataInitialized` تم إعداده إلى القيمة `true` قبل تنفيذ العمليات الإضافية على الـ `expenseData`. بالاعتماد على مدى وسع التبعيات، فإنه من الممكن أن تحتاج أيضاً إلى متغيرات مثل `isMarketingExpenseComputed`، `isSalesExpenseComputed`، وهكذا.

تُنشأ هذه التقنية متغيرات جديدة، شفرة تهيئة جديدة، وشفرة فحص خطأ جديدة، وكلها تولد إمكانيات إضافية للخطأ. يجب أن يتم أخذ بعين الاعتبار فوائد هذه التقنية ضد التعقيد الإضافي والفرصة المتزايدة لحدوث الأخطاء الثانوية، التي تولدها هذه التقنية.

2.14 العبارات البرمجية، التي ترتيبها لا يشكل مشكلة

قد تصادف حالات يبدو فيها أن ترتيب بعض التعليمات أو بضعة كتل من الشفرة غير مهم على الإطلاق. لا تعتمد تعليمة واحدة على تعليمة أخرى أو تتبعها منطقيًا. لكن الترتيب يؤثر على سهولة القراءة والأداء والصيانة، وفي غياب تبعيات ترتيب التنفيذ، يمكنك استخدام معايير ثانوية لتحديد ترتيب التعليمات أو الكتل البرمجية. المبدأ التوجيهي هو مبدأ التقارب: *حافظ على الأحداث ذات الصلة معًا*.

جعل قراءة الشفرة من الأعلى إلى الأسفل

كمبدأ عام، اجعل قراءة البرنامج من الأعلى إلى الأسفل بدلاً من القفز هنا وهناك. يُجادل الخبراء بأنه يُساهم الترتيب من الاعلى إلى الأسفل بزيادة قابلية القراءة. جعل ببساطة التحكم بالتدفق من الأعلى إلى الأسفل في

وقت التشغيل ليس بكافٍ. إذا كان على الشخص القارئ لشفرتك أن يبحث في كامل البرنامج لإيجاد المعلومات المطلوبة، فإنه يتوجب عليك أن تُعيد تنظيم الشفرة. فيما يلي مثال:

مثال بلغة البرمجة سي++ عن شفرة سيئة تقفز هنا وهناك

```
MarketingData marketingData;
SalesData salesData;
TravelData travelData;
travelData.ComputeQuarterly();
salesData.ComputeQuarterly();
marketingData.ComputeQuarterly();
salesData.ComputeAnnual();
marketingData.ComputeAnnual();
travelData.ComputeAnnual();
salesData.Print();
travelData.Print();
marketingData.Print();
```

افترض أنك ترغب بتحديد كيفية حساب marketingData. عليك أن تبدأ عند أخر سطر ومن ثم ملاحقة كل المراجع للمتغير marketingData حتى السطر الأول. يتم استخدام marketingData فقط في بضعة أماكن، ولكن عليك أن تحافظ في عقلك على كيفية استخدام marketingData في كل مكان بين المراجع الاولى والأخيرة له. بكلمات أخرى، عليك أن تنظر وتفكر في كل سطر من الشفرة في هذا المقطع لفهم كيفية حساب marketingData. وبالطبع هذا المثال هو أبسط من الشفرة، التي من الممكن أن تراها في الحياة الحقيقية. فيما يلي نفس الشفرة مع تنظيم أفضل.

مثال بلغة البرمجة سي++ لشفرة متسلسلة جيدةُ تقرأ من الأعلى إلى الأسفل

```
MarketingData marketingData;
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();
SalesData salesData;
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();
TravelData travelData;
travelData.ComputeQuarterly();
travelData.ComputeAnnual();
travelData.Print();
```

هذه الشفرة أفضل من عدة اتجاهات.[1] حيث يتم الاحتفاظ بالمراجع لكل كائن قريبة من بعضها البعض؛ إنها "موضعية". إن عدد سطور الشفرة، التي فيها الكائنات "حية"، عدد قليل. ومن الممكن أكثر أهمية، أنه الأن تبدو الشفرة كما لو أنه لا يمكن تقسيمها إلى إجرائيات منفصلة للتسويق، وللمبيعات، ولبيانات السفر. لا يُعطي أول مقطع من الشفرة تلميح بأن مثل هكذا تحليل كان ممكن.

تجميع العبارات البرمجية المرتبطة

ضع العبارات البرمجية المرتبطة مع بعضها.[2] يمكن ربطهم مع بعض لأنهم يعملون مع نفس البيانات، أو ينفذون مهمات متشابهة، أو تعتمد الواحدة منها على كون الأخريات تُنجز بترتيب.

طريقة سهلة لاختبار فيما إذا تم تجميع العبارات البرمجية المرتبطة مع بعضها البعض بطريقة جيدة، هي بطباعة قائمة من الإجرائية، ومن ثم رسم صناديق حول العبارات المرتبطة مع بعضها. إذا تم ترتيب العبارات البرمجية بطريقة جيدة، فسوف تحصل على لوحة كما هو ظاهر في الشكل 14-1، التي فيها لا تتداخل الصناديق مع بعضها البعض.



الشكل 14-1، إذا تم تنظيم الشفرة بشكل جيد في مجموعات، فلن تتداخل الصناديق المرسومة حول الأقسام المرتبطة مع بعضها. من الممكن أن تكون الصناديق متضمنه داخل بعضها البعض.

[1] **إشارة مرجعية:** المزيد حول التقنية المُعرّفة لـ "حياة" المتغيرات موجودة في "قياس زمن الحياة لمتغير" في القسم 10.4.

[2] **إشارة مرجعية:** إذا كنت تتبع عملية برمجة الشفرة الزائفة، فإن شفرتك سيتم تجميعها بشكل أتوماتيكي داخل العبارات البرمجية المرتبطة. لمزيد من التفاصيل حول هذه العملية، انظر الفصل 9، "عملية برمجة الشفرة الزائفة".

إذا لم يتم ترتيب العبارات البرمجية بشكل جيد، فسوف تحصل على لوحة مشابهة للشكل 14-2،[1] التي فيها الصناديق تتداخل مع بعضها. إذا وجدت صناديقك تتداخل، أعد تنظيم شفرتك بطريقة تصبح فيه العبارات البرمجية المرتبطة مع بعضها مُجمعة بشكل أفضل.



الشكل 14-2 إذا كانت الشفرة منظمة بشكل سيء، فسوف تتداخل الصناديق المرسومة حول الأقسام المرتبطة.

بمجرد قيامك بتجميع العبارات البرمجية المرتبطة مع بعضها البعض، فإنه يمكنك أن تجد أنهم مرتبطين مع بعضهم البعض بشكل قوي، ولا يوجد أية علاقة ذات معنى مع العبارات البرمجية التي تسبقهم أو تليهم. في مثل هكذا حالة، من الممكن أن ترغب بوضع العبارات البرمجية المرتبطة مع بعضها البعض بشكل قوي في إجرائية خاصة بها.

لائحة اختبار: تنظيم الشفرة الخطيّة2

- هل تجعل الشفرة التبعيات بين العبارات البرمجية واضحة؟
- هل تجعل أسماء الإجرائيات التبعيات واضحة؟
- هل تجعل وسطاء الإجرائيات التبعيات واضحة؟
- هل تصف التعليقات أية تبعيات من الممكن أن تكون غير واضحة؟
- هل تم ا ستخدام متغيرات التدابير التحـضيرية للتحقق من التبعيات المتـسلـسلة في الأقـسام الحرجة من الشفرة؟
- هل تمت قراءة الشفرة من الأعلى إلى الاسفل؟
- هل تم تجميع العبارات البرمجية المرتبطة مع بعضها البعض؟
- هل تم نقل مجموعات العبارات البرمجية المستقلة بشكل نسبي إلى الإجرائيات الخاصة بها؟

1 **إشارة مرجعية**: لمزيد من التفاصيل حول عمليات المحافظة على المتغيرات مع بعضها البعض، انظر القسم 4.10 "النطاق".

2 cc2e.com/1472

نقاط مفتاحية

- إن أقوى مبدأ لتنظيم الشفرة الخطيّة هو ترتيب التبعيات.
- يجب أن يتم جعل التبعيات واضــحة من خلال اســتخدام أســماء إجرائية جيدة، واســتخدام قوائم الوســطاء، واســتخدام التعليقات، و– إذا كانت الشــفرة حرجة بما فيه الكفاية- اســتخدام متغيرات التدابير التحضيرية.
- إذا لم يكن لدى الشفرة تبعيات الترتيب، حافظ على العبارات البرمجية المرتبطة قريبة من بعضها قدر الإمكان إذا كان هذا ممكن.

الفصل 15

استخدام الشَّرْطيّات

المحتويات[1]

مواضيع ذات صلة

- ترويض التعشيش العميق: القسم 19. 4
- قضايا تحكم عامة: الفصل 19
- اكتب شفرة باستخدام الحلقات: الفصل 16
- الشفرة الخطيّة: الفصل 14
- العلاقة بين أنماط البيانات وبنيان التحكم: القسم 10. 7

الشرطية هي عبارة تتحكم بتنفيذ عبارات أخرى؛ تنفيذ عبارات أخرى "مشروط" بعبارات مثل `if & else` & `case & switch`. مع أنه من المنطقي أن نشير إلى الحلقات مثل `while & for` كشرطيات أيضاً، فإننا تبعاً للعرف عالجناها بشكل منفصل. الفصل 16، "التحكم بالحلقات" سيبحث عبارتي `while & for`.

15. 1 عبارات `if`

بالاعتماد على اللغة التي تستخدمها، قد تكون قادراً على استخدام أي من الأنواع المتعددة لعبارات `if`. الأبسط هو `if` الواضحة أو عبارة `if-then`. العبارة `if-then-else` مقعدة أكثر، وسلاسل من `if-then-else-if` هي الأعقد.

[1] cc2e.com/1538

عبارات *if-then* الواضحة

اتبع هذه التوجيهات عندما تكتب عبارات `if`:



اكتب المسار الاسمي عبر الشفرة أولاً؛ ثم اكتب الحالات غير الطبيعية اكتب شفرتك بحيث يكون المسار الطبيعي عبر الشفرة واضح. تأكد من أن الحالات النادرة لا تجعل مسار التنفيذ الطبيعي غامضاً. هذا مهم لكلا سهولة القراءة والأداء.

تأكد أنك تُفرّع بشكل صحيح عند المساواة استخدام > بدلاً من >= أو < بدلاً من <= هو مشابه لصنع خطأ "بعيداً من واحدة"[1] في الدخول إلى مصفوفة أو حساب دليل حلقة. في الحلقة، فكر بنقاط النهاية لتتجنب خطأ "بعيداً من واحدة". في العبارة الشرطية، فكر بحالة "يساوي" لتتجنب ال "واحدة".

ضع الحالة الطبيعية بعد `if` بدلاً من وضعها بعد `else`[2] ضع الحالة التي تتوقعها بشكل طبيعي للمعالجة أولاً. هذا في خط المبدأ العام المتعلق بوضع الشفة التي تنتج من قرار بأقرب ما يمكن إلى القرار. هنا مثال عن شفرة تقوم بالكثير من معالجة الأخطاء، وجزافاً متفحصةً الأخطاء على طول الطريق:

مثال فيجوال بيسك عن شفرة تعالج الكثير من الأخطاء عشوائياً

[1] **مصطلحات** "خطأ بعيداً من واحدة" (`off-by-one error`) او خطأ خطوة واحدة هو النوع المنتشر بكثرة من الأخطاء البرمجية التي تحدث عندما تخطأ بشيء واحد فقط، مثل البدء بمصفوفة من 1 بدلاً من 0، فتكون أخطأت بكل شيء في ذات السياق (في المثل المضروب لن تصل إلى أي عنصر تريد في المصفوفة بل إلى الذي يليه)

[2] **إشارة مرجعية** لطرق أخرى في التعامل مع شفرة معالجة الأخطاء، انظر "ملخص لتقنيات تخفيض التعشيش العميق" في القسم 19. 4.

```
OpenFile( inputFile, status )
If ( status = Status_Error ) Then
   errorType = FileOpenError
Else
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
      SummarizeFileData( fileData, summaryData,
status )
      If ( status = Status_Error ) Then
         errorType = ErrorType_DataSummaryError
      Else
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Error ) Then
            errorType = ErrorType_SummarySaveError
         Else
            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         End If
      End If
   Else
      errorType = ErrorType_FileReadError
   End If
End If
```


هذه الشفرة صعبة التتبع لأن الحالات الطبيعية وحالات الخطأ مخلوطة جميعاً مع بعضها البعض. من الصعب إيجاد المسار الذي سيتخذ بشكل طبيعي عبر الشفرة. بالإضافة، لأن حالات الخطأ عولجت أحيانا في عبارة `if` بدلاً من `else`، من الصعب أن نكتشف أي اختبار `if` يتناغم مع الحالة الطبيعية. في الشفرة المعادة كتابتها، المسار الطبيعي بشكل مستقر شُفرت أولاً وكل حالات الخطأ شُفرت أخيراً. هذا يجعل إيجاد وقراءة الحالة الطبيعية أسهل.

مثال فيجوال ليسك عن شفرة تعالج الكثير من الأخطاء بشكل منهجي

```
OpenFile( inputFile, status )
If ( status = Status_Success ) Then
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
      SummarizeFileData( fileData, summaryData,
      status )
      If ( status = Status_Success ) Then
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Success ) Then
            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         Else
            errorType = ErrorType_SummarySaveError
         End If
      Else
         errorType = ErrorType_DataSummaryError
      End If
   Else
      errorType = ErrorType_FileReadError
   End If
Else
   errorType = ErrorType_FileOpenError
End If
```

حالة طبيعية
حالة طبيعية
حالة طبيعية
حالة طبيعية
حالة خطأ
حالة خطأ
حالة خطأ
حالة خطأ

في المثال المنقح، تستطيع أن تقرأ الدفق الرئيسي لاختبارات `if` لتجد الحالة الطبيعية. التنقيح ركز على قراءة الدفق الرئيسي بدلاً من الخوض في الحالات الاستثنائية، لذا فإن الشفرة أسهل قراءةً بعد كل شيء. مكدس حالات الخطأ في أسفل التعشيش هو علامة على شفرة معالجة الخطأ المكتوبة بشكل جيد.

هذا المثال يوضح واحداً من النهج المنظمة للتعامل مع الحالات الطبيعية وحالات الخطأ. نوقشت مجموعة من الحلول الأخرى لهذه المشكلة في هذا الكتاب، وتضمنت استخدام عبارات الحماية، والتحويل إلى إرسال متعدد الأشكال (`polymorphic dispatch`)، واستخراج القسم الداخلي لاختبار إلى إجرائية منفصلة. للحصول على لائحة كاملة بالنُهج المتاحة، راجع "ملخص تقنيات تخفيض التعشيش العميق" في القسم 19 .4.

اتبع عبارة `if` بعبارة ذات معنى أحياناً ترى شفرة تشبه المثال القادم، الذي فيه عبارة `if` هي لا-شيء:



مثال جافا عن عبارة *if* فارغة

```
if ( SomeTest )
   ;
else {
   // do something
   ...
}
```

سيتجنب معظم المبرمجين الخبراء كتابة شفرة مثل هذه،[1] كما أظن، ليتجنبوا العمل بكتابة شفرة السطر الفارغ الإضافي وسطر else. إنها تبدو سخيفة وهي تُطور بسهولة بنقض الفرض في عبارة if، ونقل الشفرة من عبارة else إلى عبارة if، والتخلص من عبارة else. إليك كيف ستبدو الشفرة بعد هذه التغيرات:

مثال جافا عن عبارة *if* فارغة مبدلة

```
if ( ! someTest ) {
   // do something
   ...
}
```

ضع بحسبانك عبارة else إذا كنت تفكر أنك تحتاج عبارة if واضحة، ضع في حسبانك إن كنت حقاً لا تحتاج عبارة if-then-else. تحليل قديم في جنرال موتورز وجد أن من 50 إلى 80 بالمئة من عبارات if ينبغي أن تتضمن عبارة else (إيلشوف 1976).

أحد الخيارات أن تشفر عبارة else-بعبارة فارغة إذا اضطررت-لتبين أن حالة else قد تم اعتبارها. تشفير elses فارغة فقط لتبين أن تلك الحالة قد تم اعتبارها يمكن أن يكون إفراط في التسلح، لكن على أقل تقدير، ضع حالة else في حساباتك. إذا كان لديك اختبار if بدون else، مالم يكن السبب واضحاً، استخدم تعليقات لتوضح عدم ضرورة عبارة else، كالتالي:

مثال جافا عن عبارة else تم التعليق المفيد عليها

```
// if color is valid
if ( COLOR_MIN <= color && color <= COLOR_MAX ) {
   // do something
   ...
}
else {
   // else color is invalid
   // screen not written to -- safely ignore command
}
```

اختبر صحة عبارة else عندما تختبر شفرتك، قد تفكر أن العبارة الرئيسية، عبارة if، هي كل ما تحتاج أن تختبر. إذا كان ممكناً أن تختبر عبارة else، على أي حال، تأكد أن تفعل ذلك.

---

1 **إشارة مرجعية** أحد مفاتيح بناء عبارة if فعالة هو كتابة التعبير المنطقي الصحيح كي تتحكم بها. لتفاصيل حول استخدام التعابير المنطقية بشكل فعال، انظر القسم 19. 1، "التعابير المنطقية"

اختبر إن كانت عبارتي if وelse مقلوبتان خطأ شائع في برمجة if-thens أن تقلب الشفرة التي يفترض أن تتبع لعبارة if والشفرة التي يفترض أن تتبع لعبارة else أو أن تعكس المنطق المستخدم في اختبار if. افحص شفرتك لتنظفها من هذا الخطأ الشائع.

سلاسل من عبارات if-then-else

في اللغات التي لا تدعم عبارات case-أو التي تدعمها جزئياً فقط-غالباً ما تجد نفسك تكتب سلاسل من اختبارات if-then-else. مثالاً، قد تستخدم شفرة تصنيف المحارف سلسلة مثل هذه:[1]

مثال لغة C++ عن استخدام سلسلة *if-then-else* لتصنيف محرف

```
if ( inputCharacter < SPACE ) {
   characterType = CharacterType_ControlCharacter;
}
else if (
   inputCharacter == ' ' ||
   inputCharacter == ',' ||
   inputCharacter == '.' ||
   inputCharacter == '!' ||
   inputCharacter == '(' ||
   inputCharacter == ')' ||
   inputCharacter == ':' ||
   inputCharacter == ';' ||
   inputCharacter == '?' ||
   inputCharacter == '-'
   ) {
   characterType = CharacterType_Punctuation;
}
else if ( '0' <= inputCharacter &&
          inputCharacter <= '9' ) {
   characterType = CharacterType_Digit;
}
else if (
   ( 'a' <= inputCharacter && inputCharacter <= 'z' )
||
   ( 'A' <= inputCharacter && inputCharacter <= 'Z' )
   ) {
   characterType = CharacterType_Letter;
}
```

ضع في حسبانك هذه التوجيهات عندما تكتب هكذا سلاسل if-then-else:

بسط الاختبارات المعقدة باستدعاءات التوابع المنطقية أحد أسباب كون الشفرة في المثال السابق صعبة القراءة هو أن الاختبارات التي تصنف المحرف معقدة. لتسهل القراءة، تستطيع استبدال استدعاءات لتوابع منطقية بهذه الاختبارات. إليك كيف تبدو شفرة المثال عندما تُستبدل التوابع المنطقية بالاختبارات:

1 إشارة مرجعية لتفاصيل أكثر حول تبسيط التعابير المعقدة، انظر القسم 19. 1، "التعابير المنطقية"

مثال لغة سي++ عن سلسلة *if-then-else* تستخدم استدعاءات التوابع المنطقية

```
if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
```

ضع الحالات الأكثر شيوعاً في البداية بوضع الحالات الأكثر شيوعاً في البداية، فإنك تقلل كمية شفرة معالجة الحالات الاستثنائية التي يتوجب على القارئ أن يقرأها حتى يجد الحالات الطبيعية. إنك تحسن الفاعلية لأنك تقلل عدد الاختبارات التي تجريها الشفرة لتجد الحالات الأكثر شيوعاً. في المثال الذي رأيته للتو، الأحرف هي أكثر شيوعاً من علامات الترقيم لكن اختبار علامات الترقيم كان في البداية. إليك الشفرة المنقحة التي تختبر الأحرف أولاً:

مثال لغة C++ عن اختبار الحالة الأكثر شيوعاً في البداية

```
if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
```

هذا الاختبار، الأكثر شيوعاً، يتم الآن في البداية

هذا الاختبار، الأقل شيوعاً، يتم الآن في النهاية

تأكد من تغطية كل الحالات اكتب عبارة else آخرة برسالة خطأ أو تأكيد لتقبض على الحالات التي لم تكن قد خططت لها. رسالة الخطأ هذه موجهة لك لا إلى المستخدم، لذا اكتبها بشكل مناسب. إليك كيف تستطيع أن تعدل مثال تصنيف المحارف كي تنجز اختبار "الحالات الأخرى":[1]

[1] **إشارة مرجعية** هذا أيضاً مثال جيد عن كيف تستطيع استخدام سلسلة اختبارات *if-then-else* بدلاً من الشفرة المعششة بعمق. لتفاصيل حول هذه التقنية، انظر القسم 19. 4، "ترويض خَطِر للتعشيش العميق".

مثال لغة C++ عن استخدام حالة الإهمال لاصطياد الأخطاء

```
if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
else {
   DisplayInternalError( "Unexpected type of
character detected." );
}
```

استبدل التركيبات الأخرى بسلاسل if-then-else **إذا كانت لغتك تدعم هذه التركيبات** لغات قليلة-مايكروسوفت فيجوال بيسك وأدا، على سبيل المثال-تدعم عبارات case التي تدعم استخدام السلاسل النصية والأنماط التعدادية والتوابع المنطقية. استخدمها-إنها أسهل قراءة وأسهل كتابة من سلاسل if-then-else. تُكتب شفرة لتصنيف أنواع المحارف باستخدام عبارة case في فيجوال بيسك كالتالي:

مثال فيجوال بيسك عن استخدام عبارة *case* بدلاً من سلسلة *if-then-else*

```
Select Case inputCharacter
   Case "a" To "z"
      characterType = CharacterType_Letter
   Case " ", ",", ".", "!", "(", ")", ":", ";", "?",
      "-"
      characterType = CharacterType_Punctuation
   Case "0" To "9"
      characterType = CharacterType_Digit
   Case FIRST_CONTROL_CHARACTER To
      LAST_CONTROL_CHARACTER
      characterType = CharacterType_Control
   Case Else
      DisplayInternalError( "Unexpected type of
                            character detected." )
End Select
```

15. 2 عبارات case

عبارة case أو switch هي تركيبة تتفاوت بشكل كبير جداً من لغة إلى لغة. C++ وجافا تدعم case فقط للأنماط الترتيبية آخذة قيمة واحدة في كل مرة. فيجوال بيسك تدعم case للأنماط الترتيبية وتمتلك قوة كتابة الاختصارات للتعبير عن مجالات ومجموعات من القيم. العديد من لغات الكتابة لا تدعم عبارة case مطلقاً.

الأقسام التالية تقدم توجيهات لاستخدام عبارات case بفعالية:

اختيار الترتيب الأكثر فعالية للحالات

تستطيع أن تختار من بين عدة طرق لتنظم الحالات في عبارة case. إذا كان لديك عبارة case صغيرة بثلاثة خيارات وثلاثة أسطر من الشفرة، الترتيب الذي تستخدمه لا يهم كثيراً. إذا كان لديك عبارة case طويلة-على سبيل المثال، عبارة case تتعامل مع دزينات من الأحداث في برنامج مُقاد بالأحداث-الترتيب مهم. فيما يلي بعض احتمالات الترتيب:

رتب الحالات أبجدياً أو رقمياً إذا كانت الحالات متساوية في الأهمية، وضعك إياها بترتيب أ-ب-ت يحسن القراءة. بهذه الطريقة يكون من السهل أن تجد حالة بين مجموعة الحالات.

ضع الحالة الطبيعية أولاً إذا كان لديك حالة طبيعية واحدة وعدة استثناءات، ضع الحالة الطبيعية أولاً. حدد بتعليقات أنها الحالة الطبيعية وأن الأخرى غير عادية.

رتب الحالات حسب التكرر ضع الحالات التي تُنفّذ عدد مرات أكثر في البداية، وذات العدد الأقل في النهاية. هذا النهج لديه حسنتان. أولاً، يستطيع الإنسان القارئ أن يجد الحالات الأكثر شيوعاً بسهولة. يكون القراء الماسحين للائحة بحثاً عن حالة محددة على الأرجح مهتمين بواحدة من الحالات الأكثر شيوعاً، ووضع الحالات الشائعة في أعلى الشفرة يجعل البحث أسرع.

نصائح لاستخدام عبارات case

هنا يوجد عدة نصائح لاستخدام عبارات case:

حافظ على بساطة أفعال كل حالة[1] حافظ على قصر الشفرة المرتبطة بكل حالة. تساعد الشفرة القصيرة التابعة لكل حالة في أن يكون بنيان عبارة case واضح. إن كانت معقدة الأفعال المنجزة لكل حالة، اكتب إجرائية واستدعي هذه الإجرائية من كل حالة بدلاً من وضع الشفرة في الحالة نفسها.

[1] **إشارة مرجعية** لنصائح أخرى حول تبسيط الشفرة، انظر الفصل 24، "إعادة التصنيع"

لا تعمل متحولات كاذبة لتجعل نفسك قادراً على استخدام عبارة case ينبغي أن تُستخدم عبارة case للبيانات البسيطة التي تُرتب بسهولة. إن لم تكن بياناتك بسيطة، استخدم سلاسل من if-then-else بدلاً. المتحولات الكاذبة مربكة، وينبغي أن تتجنبها. مثلاً، لا تقوم بهذا:




```
مثال جافا عن إنشاء متحول حالة كاذب-تطبيق سيئ
action = userCommand[ 0 ];
switch ( action ) {
   case 'c':
      Copy();
      break;
   case 'd':
      DeleteCharacter();
      break;
   case 'f':
      Format();
      break;
   case 'h':
      Help();
      break;
   ...
   default:
      HandleUserInputError(
         ErrorType.InvalidUserCommand );
}
```

المتحول الذي يتحكم بعبارة case هو action. في هذه الحالة، action أُنشئ بسلخ المحرف الأول من السلسلة userCommand، السلسلة التي أُدخلت من قبل المستخدم.

هذه الشفرة العابثة هي من الجهة الخاطئة من القرية، وهي تدعو إلى المشاكل.[1] عموماً، عندما تُصنِّع متحول كي تستخدمه في عبارة case، قد لا تدخل البيانات الحقيقية إلى عبارة case بالطريقة التي تريد. في هذا المثال، إذا كتب المستخدم copy، عبارة case تسلخ "c" الأول وبشكل صحيح تستدعي إجرائية Copy() من الناحية الأخرى، إذا كتب المستخدم cement overshoes أو clambake أو cellulite، ستسلخ عبارة case "c" أيضاً وتستدعيCopy(). اختبار الأمر الخاطئ في قسم default من عبارة case لن يعمل بشكل جيد لأنه سيتجنب فقط حروف أولى خاطئة بدلاً من أوامر خاطئة.

بدلاً من تصنيع متحولات كاذبة، ينبغي أن تستخدم هذه الشفرة سلسلة من اختبارات if-then-else-if لتتفحص السلسلة كاملةً. تبدو الشفرة بعد إعادة كتابة عفيفة كالتالي:

[1] **إشارة مرجعية** على نقيض هذه النصيحة، أحياناً يمكن أن تحسن القراءة بإسناد تعبير معقد إلى متحول منطقي مسمىً بشكل جيد أو تابع. لتفاصيل، انظر "جعل التعابير المعقدة بسيطة" في القسم 19.1

مثال جافٍ عن استخدام if-then-elses بدلاً من متحول حالة كاذب-تطبيق جيد

```
if ( UserCommand.equals( COMMAND_STRING_COPY ) ) {
   Copy();
}
else
 if ( UserCommand.equals( COMMAND_STRING_DELETE ) ) {
   DeleteCharacter();
}
else
 if ( UserCommand.equals( COMMAND_STRING_FORMAT ) ) {
   Format();
}
else
 if ( UserCommand.equals( COMMAND_STRING_HELP ) ) {
   Help();
}
...
else {
   HandleUserInputError(
      ErrorType_InvalidCommandInput );
}
```

استخدم عبارة default **"الحالة الافتراضية" (الإهمال) فقط لاكتشاف حالات الإهمال القانونية** أحياناً قد يكون لديك حالة واحدة باقية وتقرر أن تكتب شفرة تلك الحالة كعبارة افتراضية. رغم أن هذا أحياناً يكون مغرياً، فإنه غفلة (زلّة). إنك تخسر التوثيق الآلي المزود بوسومات عبارة case، وتخسر إمكانية اكتشاف الأخطاء باستخدام عبارة default. عبارات case كهذه تنهار عند التعديل.

إذا استخدمت عبارة إهمال شرعية (صحيحة)، إضافة حالة جديدة قليلة الأهمية-تضيف فقط الحالة والشفرة الموافقة لها.

إذا استخدمت عبارة إهمال مزيفة، سيكون التعديل أكثر صعوبة. عليك أن تضيف حالة جديدة، ربما جاعلاً منها عبارة الإهمال الجديدة، وثم تغير الحالة السابقة المستخدمة كحالة افتراضية (حالة إهمال) بحيث تصبح حالة شرعية. فوّق استخدم الإهمال الشرعي.

استخدم عبارة الإهمال default **لاكتشاف الأخطاء** إذا لم تُستخدم عبارة الإهمال في عبارة case لأغراض المعالجة الأخرى وليس من المفترض حدوثها، ضع رسالة تشخيص فيها:

مثال جافا عن استخدام حالة الإهمال لاكتشاف الإخطاء-تطبيق جيد

```
switch ( commandShortcutLetter ) {
   case 'a':
      PrintAnnualReport();
      break;
   case 'p':
      // no action required, but case was considered
      break;
   case 'q':
      PrintQuarterlyReport();
      break;
   case 's':
      PrintSummaryReport();
      break;
   default:
      DisplayInternalError( "Internal Error 905: Call
         customer support." );
}
```

رسائل كهذه مفيدة في كلا شفرتي التصحيح والإنتاج. معظم المستخدمين يفضلون رسالة مثل "خطأ داخلي: الرجاء الاتصال بخدمة الزبائن" لانهيار النظام، أو، أسوأ، للنتائج غير الصحيحة التي تبدو بشكل ماكر صحيحة حتى يقوم رئيس المستخدم بتدقيقها.

إذا استُخدمت عبارة الإهمال لبعض الأغراض غير اكتشاف الأخطاء، هذا يعني ضمناً أن كل حالة مختاره هي حالة صحيحة. اختبر بشكل مضاعف لتتأكد أن كل قيمة يمكن أن تدخل عبارة case ستكون شرعية. إذا أتيت ببضعة ليست شرعية، أعد كتابة العبارات بحيث تتفحص عبارة الإهمال الأخطاء.

في سي++ وجافا، تجنب الوقوع في نهاية عبارة الحالة لغة سي مشابهة ل (سي++ وجافا) لا تفصل كل حالة بشكل تلقائي. بدلاً من ذلك، يجب عليك كتابة نهاية كل حالة بشكل صريح. إذا لم تقم بتدوين نهاية الحالة، فسينخفض البرنامج خلال النهاية وينفذ الشفرة للحالة التالية.

هذا يمكن أن يؤدي إلى بعض ممارسات كتابة الشفرة الفاضحة، بما في ذلك المثال المروع التالي:



مثال سي++ عن ظلم عبارة *case*

```
switch ( InputVar ) {
   case 'A': if ( test ) {
                // statement 1
                // statement 2
   case 'B':    // statement 3
                // statement 4
                ...
                }
             ...
      break;
   ...
}
```

هذا التطبيق سيء لأنه يمزج تركيبات التحكم. تركيبات التحكم المعشقة صعبة كفاية على الفهم؛ التركيبات المتداخلة هي كل شيء ما عدا المستحيل. التعديلات على الحالة أ او الحالة ب ستكون أصعب من عمل جراحي على الدماغ، وعلى الأرجح فإن الحالات بحاجة الترتيب قبل أن يستطيع أي تعديل العمل. قد تقوم بها أيضاً بشكل صحيح في المرة الأولى. عموماً، إنها فكرة جيدة أن تتجنب الانزلاق عبر نهاية عبارة case.

في سي++، عرف بوضوح وبشكل غير قابل للخطأ معابر التدفق في نهاية كل عبارة case إذا كتبت بشكل مقصود شفرة لتنزلق عبر نهاية حالة، بوضوح اكتب تعليقاً في المكان الذي سيحدث فيه واشرح لمَ احتجت أن تكتب شفرة بهذه الطريقة.

مثال C++ عن توثيق السقوط عبر نهاية عبارة *case*

```
switch ( errorDocumentationLevel ) {
   case DocumentationLevel_Full:
      DisplayErrorDetails( errorNumber );
      // FALLTHROUGH -- Full documentation also
         prints summary comments
   case DocumentationLevel_Summary:
      DisplayErrorSummary( errorNumber );
      // FALLTHROUGH -- Summary documentation also
         prints error number
   case DocumentationLevel_NumberOnly:
      DisplayErrorNumber( errorNumber );
      break;
   default:
      DisplayInternalError( "Internal Error 905: Call
                               customer support." );
}
```

هذه التقنية مفيدة تقريباً بمقدار ما تجد شخصاً يفضل أن يقتني بونتياك أزتيك مستعملة بدلاً من كورفيتي جديدة. عموماً، الشفرة التي تسقط عبر نهاية إحدى الحالات إلى أخرى هي دعوة لارتكاب الأخطاء كلما تم تعديل الشفرة، وينبغي أن تُجتنب.

لائحة اختبار: استخدام الشرطيات1

عبارات if-then

- هل المسار الإسمي عبر الشفرة واضح؟
- هل اختبارات if-then تتفرع بشكل صحيح عند المساواة؟
- هل عبارة else حاضرة وموثقة؟
- هل عبارة else صحيحة؟
- هل استُخدمت عبارتي if وelse بشكل صحيح-ليستا معكوستين؟
- هل الحالة الطبيعية تتبع if بدلاً من else؟

[1] cc2e.com/1545

سلاسل `if-then-else`

- هل تم تغليف الاختبارات المعقدة في استدعاءات توابع منطقية؟
- هل تم اختبار الحالة الأكثر شيوعاً في البداية؟
- هل تمت تغطية كل الحالات؟
- هل سلسلة if-then-else-if هي التطبيق الأفضل-أفضل من عبارة case
- عبارات case
- هل تم ترتيب الحالات معنوياً؟
- هل أفعال كل حالة بسيطة-تستدعي الإجرائيات الأخرى عند الضرورة؟
- هل تختبر عبارة case متحولاً حقيقاً، لا كاذباً أنشئ فقط ليستخدم ويظلم عبارة case؟
- هل استخدام عبارة الإهمال شرعي؟
- هل استُخدمت عبارة الإهمال لاكتشاف وتدوين الحالات غير المتوقعة؟
- هل انتهت كل حالة ب break في سي أو سي++ أو جافا؟

نقاط مفتاحية

- انتبه إلى ترتيب عبارتي if وelse في عبارات if-else البسـيطة، خصـوصـاً إذا كانت تعالج الكثير من الأخطاء. تأكد من أن الحالة الإسمية واضحة.
- اختر ترتيباً يعظّم سهولة القراءة في عبارات case وسلاسل if-then-else.
- استخدم عبارة الإهمال في عبارة case أو آخر else في سلسلة عبارات if-then-else، لتصطاد الأخطاء.
- لم تُخلق تركيبات التحكم متساوية. اختر تركيبة التحكم التي تناسب أكثر لكل قسم من الشفرة.

الفصل 16

التحكم بالحلقات

المحتويات[1]

مواضيع ذات صلة

- ترويض التداخل العميق: القسم 4.19
- قضايا التحكم العامة: الفصل 19
- استخدام الشرطيّات: الفصل 15
- الشفرة الخطيّة: الفصل 14
- العلاقة بين أنواع البيانات وهياكل التحكم: الفصل 7.10

"الحلقة" هي مصطلح غير رسمي يُشير لأي نوع من بُنى التحكم التكرارية- أي بنية تؤدي بالبرنامج لتنفيذ كتلة من الشفرات بشكل متكرر. نماذج الحلقات الشائعة هي: `for`، و`while`، و`do-while` بلغة سي++ وجافا، وكذلك `for-next`، و`while-wind`، و`Do-loop-while` بلغة مايكروسوفت فيجوال بيسيك. إن استخدام الحلقات هو أحد أكثر مظاهر البرمجة تعقيدًا، حيث أن معرفة كيف ومتى تستخدم كل نوع من الحلقات عامل حاسم في بناء برمجة عالية الكفاءة.

1.16 اختيار نوع الحلقة

في أغلب اللغات، ستستخدم بعض الأنواع من الحلقات:

- الحلقة المعدودة: تُنجز عدد محدد من المرات، على سبيل المثال مرة لكل موظف.

[1] cc2e.com/1609

- الحلقة المقيمة باستمرار: لا تعرف مسبقًا كم مرة ستنفذ وتختبر إن كانت قد انتهت في كل تكرار. مثلًا، ستبقى تعمل طالما يوجد نقود، حتى يختار المستخدم الخروج، أو حتى تصادف خطأ ما.
- الحلقة اللانهاية: تنفذ للأبد حالما تبدأ. إنها النوع الذي تجده في الأنظمة المدمجة مثل منظم ضـــربات القلب، وأفران المايكروويف، وأنظمة الملاحة.
- حلقة التكرار: تنجز عملها مرة لكل عنصر في الصف الحاوي.

تختلف أنواع الحلقات فيما بينها أولا بمرونتها– إن كانت الحلقة تنفذ عدد محدد من المرات أو إن كانت تختبر اكتمال الحلقة في كل تكرار.

وكذلك تختلف أنواع الحلقات بموقع اختبار الاكتمال. بوسعك وضع الاختبار في بداية أو وسط أو نهاية الحلقة. تخبرك هذه الخاصية إن كانت الحلقة تُنفذ مرة واحدة على الأقل. إذا اختبرت الحلقة في البداية، فليس بالضرورة أن ينفذ جسم الحلقة، وإذا اختبرت الحلقة في النهاية، فإن جسم الحلقة سينفذ مرة واحدة على الأقل. وإذا اختبرت الحلقة في الوسط، جزء الحلقة الذي يسبق الاختبار سينفذ مرة واحدة على الأقل، ولكن جزء الحلقة الذي يلي الاختبار فليس بالضرورة أن ينفذ مطلقًا.

تقرر مرونة وموقع الاختبار أي نوع حلقات ستُختار كبنية تحكم. الجدول 16-1 يبين أنواع الحلقات بعدة لغات ويصف مرونة وموقع اختبار كل حلقة.

جدول 16-1 أنواع الحلقات:

اللغة	نوع الحلقة	المرونة	موقع الاختبار
فيجول بيسك	For-Next	غير مرنة	البداية
	While-Wend	مرنة	البداية
	Do-Loop-While	مرنة	البداية أو النهاية
	For-Each	غير مرنة	البداية
سي/سي ++/ سي#/جافا	For	مرنة	البداية
	While	مرنة	البداية
	do-while	مرنة	النهاية
	*foreach	غير مرنة	البداية

متى تستخدم حلقة while

يعتقد المبرمجون المبتدئون أحيانًا بأن حلقة while مستمرة التقييم وهكذا هي تنتهي لحظة عدم تحقق الشرط، بغض النظر عن أيّة عبارة يتم تنفيذها في الحلقة (كورتيس وآخرون 1986). بالرغم من أن هذا ليس مرن تمامًا، اختيار حلقة while هو اختيار لحلقة مرنة. إذا لم تكن تعرف بشكل مُسبق كم مرة بالضبط تريد أن تتكرر الحلقة، استخدم حلقة while. على عكس ما يفكر فيه بعض الملاحظين، يُنفذ اختبار الخروج من الحلقة

فقط مرة واحدة في كل دورة للحلقة، والقضية الرئيسية بالنسبة لحلقات while هو اتخاذ قرار موقع الاختبار عند بداية أو عند نهاية الحلقة.

الحلقات مع الاختبار في البداية

من أجل الحلقة التي تختبر في البداية، بإمكانك استخدام حلقة while في لغة سي++، وسي#، وجافا، وفيجوال بيسك، ومعظم اللغات الأخرى. ويمكنك محاكاة حلقة while في اللغات الأخرى.

الحلقات مع الاختبار في النهاية

ربما سيكون لديك بين الفينة والأخرى حالة تريد فيها حلقة مرنة، ولكن الحلقة يجب أن تنفذ مرة واحدة على الأقل. في مثل هذه الحالة، بإمكانك استخدام حلقة while التي تنجز الاختبار في نهايتها. بإمكانك استخدام حلقة do-while بلغة سي++ ولغة سي# ولغة جافا، وحلقة do-loop-while بلغة فيجول بيسك، ومعظم اللغات الأخرى. ويمكنك محاكاة الحلقات التي تختبر في النهاية في أي لغة أخرى.

متى تستخدم حلقة (حلقة مع Exit (Loop-With-Exit

حلقة Loop-With-Exit هي حلقة فيها يظهر شرط الخروج في وسط الحلقة بدلًا من ظهوره في البداية أو في النهاية. حلقة Loop-With-Exit متوفرة على نحو واضح بلغة فيجول بيسك، ويمكنك أن تحاكي حلقة Loop-With-Exit مع البنية الهيكلية لحلقة while وbreak في لغة سي++، ولغة سي ولغة جافا، أو مع تعليمات goto بأي لغة أخرى.

حلقات Loop-With-Exit العادية

تتألف حلقة Loop-With-Exit عادةً من بداية الحلقة، جسم الحلقة (متضمنة شرط خروج)، ونهاية الحلقة، كما في مثال فيجول بيسك التالي:

مثال بلغة البرمجة فيجوال بيسك لحلقة (حلقة مع Exit) عامة



الاستخدام النموذجي لحلقة loop-with-exit هو للحالة التي يحتاج فيها الاختبار في البداية أو في النهاية إلى حلقة ونصف(loop-and-a-half). هنا مثال بلغة سي ++ للحالة التي تبرر حلقة loop-with-exit، ولكن لا تستخدمها.

```
مثال بلغة البرمجة سي++بشفرة مُضاعفة سوف تنهار عند الصيانة.
// حساب الدرجات و التقييمات.
score = 0;
GetNextRating( &ratingIncrement );
rating = rating + ratingIncrement;
while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
 GetNextScore( &scoreIncrement );
 score = score + scoreIncrement;
 GetNextRating( &ratingIncrement );
 rating = rating + ratingIncrement;
}
```

تظهر هذه السطور هنا....

...وتُكرر هنا.

سطري الشفرة في بداية المثال مكررين في آخر سطرين من الشفرة من حلقة while. خلال التعديل، بإمكانك بسهولة نسيان المحافظة على مكان السطرين المتوازيين. قد لا يدرك مبرمج آخر يعدل الشفرة بأنه من المفترض أن يعدل مكاني السطرين بشكل متوازي. وإلا ستكون في النتيجة أخطاء متزايدة من التعديلات غير المكتملة. هنا نبين كيف يمكنك كتابة الشفرة بشكل أوضح:

مثال بلغة البرمجة فيجوال بيسك لحلقة (حلقة مع Exit) أسهل صيانتها

```
//حساب الدرجات و التقييمات.تستخدم الشفرة حلقة لانهائية//
//وعبارات كسر للحلقة لمحاكاة الحلقة مع exit.
score = 0;
while ( true ) {
 GetNextRating( &ratingIncrement );
 rating = rating + ratingIncrement;
 if ( !( ( score < targetScore ) && ( ratingIncrement != 0 ) ) ) {
 break;
 }
 GetNextScore( &scoreIncrement );
 score = score + scoreIncrement;
}
```

هذا شرط الخروج من الحلقة (والآن يمكن أن تُبسط باستخدام نظرية ديمورغان، المشروحة في القسم 19.1).

وهنا كيف تكتب نفس الشفرة باستخدام فيجول بيسيك:

مثال بلغة البرمجة فيجوال بيسك لحلقة (حلقة مع Exit)

```
' حساب الدرجات و التقييمات
score = 0
Do
 GetNextRating( ratingIncrement )
 rating = rating + ratingIncrement
 If ( not ( score < targetScore and ratingIncrement <> 0 ) ) Then Exit
Do
 GetNextScore( ScoreIncrement )
 score = score + scoreIncrement
Loop
```

ضع بالاعتبار هذه النقاط الحساسة عندما تستخدم هذا النوع من الحلقات:

ضع كل شروط الخروج من الحلقة في مكان واحد.[1] يضمن توزيعها عمليًا إغفال شرط أو آخر خلال التصحيح أو التعديل أو الاختبار.

استخدم التعليقات للإيضاح. إذا استخدمت تقنية حلقة loop-with-exit بلغة لا تدعمها مباشرةً، استخدم التعليقات لجعل ما تقوم به واضحًا.

لدى حلقة loop-with-exit مدخل واحد ومخرج واحد، وبنية تحكم هيكلية، وهي النوع المفضل من حلقات التحكم (نادي البرمجة الإنتاجية 1989)، وقد بينت بأنها الأسهل للفهم أكثر من أي نوع حلقات آخر. قارنت دراسة لطلاب البرمجة هذا النوع من الحلقات مع تلك التي تنتهي إما بالأعلى أو الأسفل (سولواي، بونر، وايرليتش 1983).حصل الطلاب على 25 بالمئة أعلى في اختبار الإدراك عندما استخدمت حلقات loop-with-exit، وانتهى كاتب الدراسة بأن هيكلية حلقة loop-with-exit تجسد الطريقة التي يفكر بها الناس حول التحكم بالتكرار بصورة أقرب مما تفعله بُنى الحلقات الأخرى.

في الممارسات العملية الشائعة، حلقة loop-with-exit ليست مستخدمة بشكل كبير حتى الآن. لا يزال المحلفون في الغرفة الدخانية يتناقشون فيما إذا كانت هذه الحلقة ممارسة جيدة لإنتاج الشفرة. طالما أن المحلفون لا يزالون في الداخل، فإن حلقة loop-with-exit تقنية جيدة لوضعها في صندوق أدواتك البرمجية– ما دمت تستخدمها بحذر.

حلقات Loop-With-Exit الشاذة

[1] **إشارة مرجعية**: المزيد من التفاصيل عن شروط الخروج معروض لاحقًا في هذا الفصل. لتفاصيل عن استخدام التعليقات مع الحلقات، انظر "التعليق على هيكلة التحكم" في القسم 5.32.

نوع آخر من حلقات Loop-With-Exit هي تلك المستخدمة لتجنب حلقة loop-and-a-half كما يظهر هنا:

مثال بلغة البرمجة سي++لوصول إلى منتصف الحلقة باستخدام goto – ممارسة عملية سيئة

شفرة مرعبة

```
goto Start;
while ( expression ) {
 // do something
 ...
 Start:
 // do something else
 ...
}
```

بنظرة أولية، هذا يبدو مشابهًا لأمثلة حلقة loop-with-exit السابقة. إنها المستخدمة في المحاكاة التي فيها // do something لا تحتاج لأن تنفذ في المسار الأول عبر الحلقة ولكن // do something else ينفذ. إنها بنية تحكم دخل واحد وخرج واحد: الطريقة الوحيدة لدخول الحلقة هي الأمر goto في الأعلى، والطريقة الوحيدة للخروج منها هي عبر اختبار while. في هذا النهج مشكلتين: إنه يستخدم الأمر goto وهو غريب كفاية حتى يكون مزعجًا.

يمكنك إنجاز نفس التأثير بلغة سي ++ بدون استخدام goto، كما هو موضح في المثال التالي. إذا كانت اللغة التي تستخدمها لا تدعم الأمر break، تستطيع محاكاته بالأمر goto.

مثال بلغة البرمجة سي++ لشفرة معاد كتابتها بدون استخدام goto – ممارسة عملية أفضل

```
while ( true ) {
 // do something else
 ...
 if ( !( expression ) ) {
 break;
 }
 // do something
 ...
}
```

بُدلت الكتل البرمجية قا. وبعد break.

متى تُستخدم حلقة for

إن حلقة for خيار جيد عندما تريد حلقة تُنفذ عدد محدد من المرات.[1] يمكنك استخدام حلقة for بلغة سي ++ وسي# وجافا وفيجوال بيسك، ومعظم اللغات الأخرى.

[1] **قراءة متعمقة:** لمزيد من الإرشادات الجيدة حول استخدام حلقات for انظر " كتابة الشفرات الصلبة (ماغواير 1993)

استخدم حلقات for للنشاطات البسيطة التي لا تتطلب أي تحكم داخلي في الحلقة. استخدمها عندما تتضمن حلقة التحكم تزايد بسيط أو تناقص بسيط، كالمرور على العناصر في حاوية. النقطة في حلقة for إنك تقوم بتهيئتها في أعلى الحلقة وبعدها إنس أمرها. ليس عليك القيام بأي شيء داخل الحلقة للتحكم بها. إذا كان لديك شرط يجب أن يقفز التنفيذ بموجبه خارج الحلقة، استخدم حلقة while بدلًا منها.

وبالمثل، لا تغير بشكل صريح قيمة المُدخل لحلقة for لإجبارها على الانتهاء. استخدم عندها حلقة while بدلًا منها. إن حلقة for للاستخدامات البسيطة. معظم مهام الحلقات المعقدة من الأفضل التعامل معها بحلقة while.

متى نستخدم حلقة foreach

حلقة foreach أو مكافئها (foreach بلغة السي #، For-Each بلغة فيجوال بيسك، for-in بلغة بايثون) هي مفيدة لإنجاز عملية على كل عنصر من مصفوفة أو حاوي آخر. لديها حسنه بالقضاء على العمليات الحسابية في الحلقة وبهذا إلغاء أية فرصة لارتكاب خطأ فيها. فيما يلي مثال عن هذا النوع من الحلقات:

مثال بلغة البرمجة سي# لاستخدام الحلقة foreach.

```
int [] fibonacciSequence = new int [] { 034 ،21 ،13 ،8 ،5 ،3 ،2 ،1 ،1 ، };
int oddFibonacciNumbers = 0;
int evenFibonacciNumbers = 0;
// حساب عدد الأعداد الفردية و الزوجية في المصفوفة fibonacciSequence
foreach ( int fibonacciNumber in fibonacciSequence ) {
 if ( fibonacciNumber % 2 ) == 0 ) {
 evenFibonacciNumbers++;
 }
 else {
 oddFibonacciNumbers++;
 }
}
Console.WriteLine( "Found {0} odd numbers and {1} even numbers." ،
 oddFibonacciNumbers ،evenFibonacciNumbers );
```

2.16 التحكم بالحلقة

ما الخطأ الذي من الممكن أن يحدث مع حلقة؟ ستتضمن أية إجابة على هذا السؤال التهيئة غير الصحيحة أو المحذوفة للحلقة، أو تهيئة محذوفة للمراكم "عداد الحلقة" أو للمتحولات الأخرى المتعلقة بالحلقة، أو نسيان

زيادة قيمة متحول الحلقة أو زيادة المتحول بشكل غير صحيح، أو فهرسة عناصر مصفوفة من مُدخلات حلقة بشكل غير صحيح.



يمكنك منع هذه المشاكل بمراقبة اثنين من الممارسات. أولًا، تقليل إلى الحد الأدنى لعدد العوامل التي تؤثر بالحلقة. بسّط! بسّط! بسّط! ثانيًا، عامل داخل الحلقة كأنه إجرائية- احتفظ بالقدر الممكن من التحكم بالحلقة خارجها. بيّن بشكل صريح الشروط التي يتم فيها تنفيذ جسم الحلقة. لا تسمح للقارئ بأن ينظر داخل الحلقة ليفهم تحكم الحلقة. فكر بالحلقة كصندوق أسود: البرنامج المحيط يعرف شروط التحكم ولكن ليس المحتويات.[1]

مثال بلغة البرمجة سي++ عن التعامل مع الحلقة كصندوق أسود.

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {
 ████████████████████████████
}
```

ما هي الشروط التي تنتهي بها الحلقة؟ بشكل واضح، كل ما تعرفه أنه إما `inputFile.EndOfFile()`يصبح صحيح أو `MoreDataAvailable` يصبح خطأ.

دخول الحلقة

استخدم هذه التعليمات عند الدخول بالحلقة:

ادخل الحلقة من موقع واحد فقط. يسمح لك تنوع بنى تحكم الحلقة بالاختبار في البداية أو في الوسط أوفي نهاية الحلقة. هذه البنية غنية بما يكفي للسماح لك بالدخول للحلقة من الأعلى دائمًا. ليس عليك الدخول من أماكن متعددة.

ضع لشفرة التهيئة مباشرةً قبل الحلقة. يدافع مبدأ التقريب عن وضع العبارات المتعلقة ببعضها معًا. إذا كانت العبارات المتعلقة ببعضها متناثرة عبر إجرائية، فمن السهل إهمالها خلال التعديل والقيام بتعديلات خاطئة. أما إذا كانت العبارات المتعلقة بعضها محفوظة معًا، فمن السهل تجنب الأخطاء خلال التعديل.

اجعل عبارات تهيئة حلقة مع الحلقة المتعلقة بها[2]. إن لم تفعل هذا، فإنك أكثر عرضة لاقتراف أخطاء عندما تعمم الحلقة إلى حلقة أكبر وتنسى تعديل شفرة التهيئة. نفس النوع من الأخطاء ممكن الحدوث عندما تحرك أو

---

[1] **إشارة مرجعية**: إذا استخدمت تقنية حلقة `while(true)-break` الموصوفة فيما سبق. شرط الخروج يكون داخل الصندوق الأسود، حتى لو استخدمت شرط خروج وحيد فقط، فستخسر فائدة التعامل مع الحلقة كصندوق أسود.

2 **إشارة مرجعية**: للمزيد عن تحديد مجال متغيرات الحلقة، انظر " حدود مجال متغيرات فهرس الحلقة بالنسبة للحلقة نفسها" فيما بعد ضمن هذا الفصل.

تنسخ شفرة الحلقة إلى إجرائيات مختلفة دون تحريك أو نسخ شفرتها التمهيدية. وضع التهيئة الأولية بعيداً عن الحلقة – في قسم تعريف البيانات أو قسم تدبير الأمور في أعلى الإجرائية التي تحوي الحلقة- يستدعي مشاكل التهيئة.

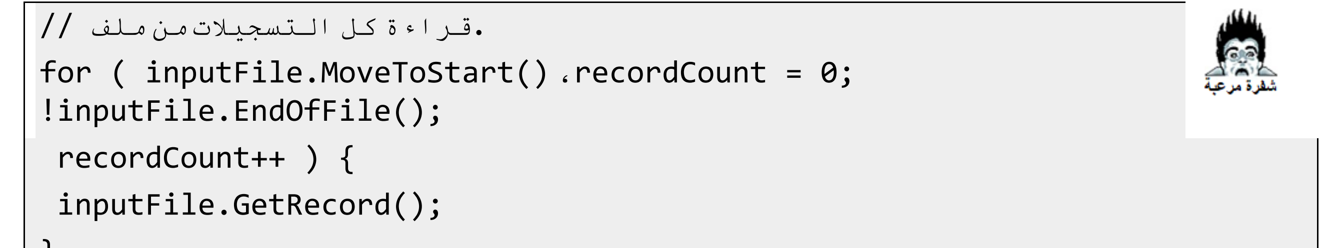
استخدم حلقة `while (true)` ***للحلقات اللانهائية***. قد يكون لديك حلقة تعمل بدون نهاية – مثلًا، حلقة منظم دقات القلب أو فرن الموجة الميكروية. أو قد يكون لديك حلقة تنتهي فقط في حالة استجابتها لحدث ما – "حلقة الحدث". يمكن تشفير حلقات لانهائية بعدة طرق. تزييف حلقة لانهائية بعبارات مثل `for i=1` إلى 99999 هو خيار سيئ لأن حدود الحلقة المحددة تلوث محتوى الحلقة – 99999 قد يكون قيمة حقيقية. من الممكن أن تنهار هذه الحلقات اللانهائية المزيفة أيضًا عند الصيانة.

يعتبر مصطلح حلقة `while (true)` طريقة قياسية لكتابة حلقة لا نهائية بلغة سي ++ وجافا وفيجوال بيسك واللغات الأخرى التي تدعم هيكليات قابلة للمقارنة. يفضل بعض المبرمجين استخدام حلقة `for(;;)`، والتي تقدم بديل مقبول.

فضّل حلقات *for* ***عندما يكون استخدامها مناسب.*** تجّمع حلقة `for` شفرة تحكم الحلقة في مكان واحد، والذي يجعل قابلة قراءة الحلقات أسهل. يحدث أحد الأخطاء الشائعة للمبرمجين عند تعديل البرمجية بتغيير شفرة تهيئة الحلقة في أعلى حلقة `while` ولكن يُنسى تغيير الشفرة المتعلقة بالتهيئة في أسفل الحلقة. في حلقة `for`، تُجمّع معًا كل الشفرات المناسبة في أعلى الحلقة، الأمر الذي يجعل تصحيح التعديلات أسهل. إذا استطعت استخدام حلقة `for` بشكل مناسب بدلاً من أي نوع آخر من الحلقات، فافعل ذلك.

لا تستخدم حلقة *for* ***عندما تكون حلقة*** *while* ***مناسبة أكثر.*** إن الإساءة الشائعة لمرونة هيكلية حلقة `for` بلغة سي ++ وسي # وجافا هي حشو محتوى خلية `while` بشكل عشوائي في رأس حلقة `for`. يُظهر المثال التالي حلقة `while` محشوة في رأس حلقة `for`.

مثال بلغة البرمجة سي++ عن حلقة while محشورة بشكل سيء في رأس حلقة for.



```
// قراءة كل التسجيلات من ملف.
for ( inputFile.MoveToStart() ،recordCount = 0;
!inputFile.EndOfFile();
 recordCount++ ) {
 inputFile.GetRecord();
}
```

فائدة حلقة for في لغة البرمجة سي ++ أكثر من فوائدها بباقي اللغات كونها أكثر مرونة من جهة التهيئة ومعلومات الانتهاء التي يمكنها استخدامها. الضعف المتأصل في هكذا مرونة هو أنك تستطيع وضع عبارات في رأس الحلقة لا تقوم بأي شيء من التحكم بالحلقة.

احجز رأس حلقة for لعبارات التحكم بالحلقة– العبارات التي تهيئ الحلقة، أو تنهيها، أو تحركها باتجاه النهاية. في المثال السابق، تحرك عبارات inputFile.GetRecord() في جسم الحلقة باتجاه النهاية، ولكن عبارات recordCount لا تفعل ذلك، إنها فقط عبارات "تدابير تحضيرية" (housekeeping) لا تتحكم بتقدم الحلقة. إن وضع العبارات recordCount في رأس الحلقة وترك عبارة inputFile.GetRecord() خارجاً أمر مضلل، حيث تخلق انطباع كاذب بأن recordCount يتحكم بالحلقة.

إذا أردت استخدام حلقة for أكثر من حلقة while في هذه الحالة، ضع عبارات التحكم بالحلقة في رأس الحلقة ودع كل شيء آخر خارجًا. هنا الطريقة الصحيحة لاستخدام رأس الحلقة:

مثال بلغة البرمجة سي++ إذا كان الاستخدام غير تقليدي لرأس الحلقة for.

```
recordCount = 0;
for ( inputFile.MoveToStart(); !inputFile.EndOfFile();
inputFile.GetRecord() ) {
 recordCount++;
}
```

محتويات رأس الحلقة في هذا المثال كلها متعلقة بالتحكم بالحلقة. العبارة inputFile.MoveToStart() تهيئ الحلقة. تختبر العبارة !inputFile.EndOfFile() فيما إذا كانت الحلقة انتهت، وتحرك العبارة inputFile.GetRecord() الحلقة باتجاه النهاية. لا تحرك العبارات التي تؤثر على recordCount الحلقة مباشرة للانتهاء وغير مضمنة في رأس الحلقة بشكل مباشر. لا تزال الحلقة while على الأرجح أكثر ملائمة لهذا العمل، ولكن على الأقل تستخدم هذه الشفرة رأس الحلقة بشكل منطقي. من أجل السجل، هنا كيف تبدو الشفرة عندما تستخدم حلقة while:

مثال بلغة البرمجة سي++ للاستخدام المناسب لحلقة while.

```
//قراءة التسجيلات من ملف.
inputFile.MoveToStart();
recordCount = 0;
while ( !inputFile.EndOfFile() ) {
 inputFile.GetRecord();
 recordCount++;
}
```

معالجة وسط الحلقة

يصف الجزء التالي كيفية التعامل مع وسط الحلقة:

استخدم القوسين { و} لحصر العبارات في الحلقة. استخدم أقواس الشفرة في كل مرة. إنها لا تكلف شيئًا من سرعة أو مساحة وقت التشغيل، بل إنها تساعد على تسهيل القراءة، وتساعد على منع الأخطاء عند تعديل الشفرة. إنها ممارسة وقائية برمجية جيدة.

تجنّب الحلقات الفارغة. في لغة سي ++ وجافا، من الممكن إنشاء حلقة فارغة، تلك التي يكون فيها عمل الحلقة مشفرًا في نفس سطر الاختبار الذي يفحص فيما إذا كان العمل قد انتهى. هنا مثال على ذلك:

مثال بلغة البرمجة سي++ لحلقة فارغة.

```
while ( ( inputChar = dataFile.GetChar() ) != CharType_Eof ) {
 ;
}
```

في هذا المثال، الحلقة فارغة لأن تعبير حلقة while يتضمن شيئين: عمل الحلقة - inputChar = () dataFile.GetChar– وفحص إذا كانت الحلقة ستنتهي - inputChar != CharType_Eof -. ستكون الحلقة أوضح إذا أعيدت كتابتها بحيث يكون العمل الذي تقوم به جليّ للقارئ:

مثال بلغة البرمجة سي++ لحلقة فارغة محولة إلى حلقة مشغولة.

```
do {
 inputChar = dataFile.GetChar();
} while ( inputChar != CharType_Eof );
```

تأخذ الشفرة الجديدة ثلاث أسطر كاملة بدلًا من سطر واحد وفاصلة منقوطة، وهي مناسبة إذ أنها تعمل عمل ثلاثة أسطر بدل العمل المرتبط بسطر واحد وفاصلة منقوطة.

حافظ على مهمة تدبير الأمور التحضيرية (*Loop-housekeeping chores* ***) الحلقة إما في بداية أو نهاية الحلقة***. مهمة تدبير أمور الحلقة هي تعابير مثل i=i+1 أو j++، التعابير التي هدفها الرئيسي ليس إنجاز عمل الحلقة بل التحكم بالحلقة. تم تدبير أمور الحلقة في نهاية الحلقة في هذا المثال:

مثال بلغة البرمجة سي++ عن عبارات تدبير أمور في نهاية الحلقة.

```
nameCount = 0;
totalLength = 0;
while ( !inputFile.EndOfFile() ) {
 // القيام بعمل الحلقة.
```

```
 inputFile >> inputString;
 names[ nameCount ] = inputString;
 ...
 // الاستعداد إلى المرور الثاني من خلال تدابير الحلقة.
 nameCount++;
 totalLength = totalLength + inputString.length();
}
```

هنا تعابير التدابير للحلقة.

كقاعدة عامة، المتغيرات التي هيئتها قبل الحلقة هي المتغيرات التي يتعامل معها في جزء تدبير الأمور من الحلقة.

دع كل حلقة تنجز تابع واحد فقط[1]. الحقيقة المجردة بأن الحلقة تستطيع أن تُستخدم لإنجاز عملين في نفس الوقت ليست تبرير كافي لفعلهم سويًا. يجب أن تكون الحلقات كالإجرائيات، حيث أن كل واحدة يجب أن تُنجز عملاً واحدًا وأن تنجزه بشكل جيد. إذا بدا استخدام حلقتين بأنه غير فعال عندما يكون استخدام واحدة يفي بالغرض، اكتب الشفرة كحلقتين، ضع تعليق بأنه يمكن أن تجميعهم في حلقة واحدة لزيادة الفعالية، وعندها انتظر حتى تُظهر اختبارات الأداء بأن هذا القسم من البرنامج يخلق مشاكل أداء قبل تغيير الحلقتين لواحدة.

الخروج من الحلقة

يصف هذا قسم التعامل مع نهاية الحلقة:

أكد لنفسك بأن الحلقة ستنتهي. هذا الأمر أساسي، حاكي ذهنيًا تنفيذ الحلقة حتى تتأكد، في كافة الظروف، أنها تنتهي. فكر من خلال الحالات الاسمية، ونقاط النهاية، وكل الحالات الاستثنائية.

اجعل شروط انتهاء الحلقة واضحة. إذا استخدمت حلقة for ولم تنخدع بفهرس الحلقة، ولم تستخدم goto أو break للخروج من الحلقة، ستكون شروط الانتهاء واضحة. وبالمثل، إذا كنت تستخدم حلقة while أو حلقة repeat-until وتضع كل التحكم في عبارة while أو repeat-until، فإن شرط الانتهاء سيكون واضح. المفتاح هو وضع التحكم في مكان واحد.

لا تعبث مع فهرس حلقة for لجعل الحلقة تنتهي. يُغير بعض المبرمجين عنوةً قيمة فهرس حلقة for لجعل الحلقة تنتهي باكرًا. هنا مثال على ذلك:

مثال جافا عن العبث بفهرس الحلقة.

شفرة مرعبة

```
for ( int i = 0; i < 100; i++ ) {
 // some code
```

[1] **إشارة مرجعية:** للمزيد من التحسين انظر الفصل 25 " استراتيجيات انسجام الشفرة" والفصل 26 " تقنيات انسجام الشفرة ".

```
 ...
 if ( ... ) {
 i = 100;
 }
 // more code
 ...
}
```

هنا العبث

القصد من هذا المثال هو إنهاء الحلقة تحت بعض الشروط بتغيير قيمة i إلى 100، القيمة التي هي أكبر من مجال نهاية حلقة for من 0 إلى 99. عمليًا يتجنّب كل المبرمجين هذه الممارسة، إنها دلالة على مُبرمج هاوي. عندما تهيئ حلقة for، يكون عداد الحلقة خارج الحدود. استخدم حلقة while لتوفير المزيد من التحكم على شروط الخروج من الحلقة.

تجنّب الشفرة التي تعتمد على قيمة فهرس الحلقة النهائية. إنه لنموذج سيئ أن تستخدم قيمة فهرس الحلقة بعد الحلقة. تختلف القيمة النهائية لفهرس الحلقة من لغة إلى أخرى ومن تنفيذ لتنفيذ. تختلف القيمة عندما تنتهي الحلقة بشكل عادي وعندما تنتهي بشكل شاذ. حتى لو صادف أنك تعرف القيمة النهائية دون التوقف للتفكير بها، فمن المحتمل أن يفكر الشخص التالي الذي سيقرأ الشفرة بها. إن الصيغة الأفضل والأكثر توثيقًا بشكل ذاتي هي إسناد القيمة النهائية لمتحول في النقطة المناسبة داخل الحلقة.

تسيء هذه الشفرة استخدام قيمة الفهرس النهائية:

مثال بلغة البرمجة سي++ عن شفرة تُسيء استخدام القيمة النهائية لفهرس الحلقة.

```
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
 if ( entry[ recordCount ] == testValue ) {
 break;
 }
}
// lots of code
...
if ( recordCount < MAX_RECORDS ) {
 return( true );
}
else {
 return( false );
}
```

هنا اساءة استخدام القيمة الطرفية (النهائية) لفهرس

في هذا الجزء، يجعل الاختبار الثاني لـ recordCount<Max-Record يظهر بأن الحلقة يُفترض أن تبحث خلال كل القيم التي في []entry وتعيد القيمة true إذا وجدت المكافئ لقيمة الاختبار وfalse إن لم تجده.

من الصعب التذكر فيما إذا كان الفهرس قد ازداد قبل نهاية الحلقة، لذا من السهل ارتكاب خطأ الخطوة الواحدة. من الأفضل لك كتابة الشفرة التي لا تعتمد على القيمة النهائية للفهرس. هنا كيف تُكتب الشفرة:

مثال بلغة البرمجة سي++ عن شفرة لا تُسيء استخدام القيمة النهائية لفهرس الحلقة.

```
found = false;
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
 if ( entry[ recordCount ] == testValue ) {
 found = true;
 break;
 }
}
// lots of code
...
return( found );
```

يستخدم هذا الجزء الثاني من الشفرة متغير زيادة ويحافظ على مرجعية recordCount أكثر مرجعية. كما هي الحالة غالبًا عند استخدام متغير منطقي إضافي، شفرة النتائج تكون أكثر وضوحًا.

خذ بعين الاعتبار استخدام عدادات سلامة. عداد السلامة هو متغير تزيده كل مرور عبر الحلقة للانتهاء إذا كانت الحلقة قد تم تنفيذها مرات كثيرة. إذا كان لديك برنامج يكون الخطأ فيه فادحًا، بإمكانك استخدام عدادات سلامة للتأكد من انتهاء الحلقات. هذه الحلقة بلغة سي ++ تستطيع بشكل مريح استخدام عداد سلامة:

مثال بلغة البرمجة سي++ عن حلقة تستطيع استخدام عداد سلامة.

```
do {
 node = node->Next;
 ...
} while ( node->Next != NULL );
```

هنا نفس الشفرة مع عدادات سلامة مُضافة:

مثال بلغة البرمجة سي++ عن استخدام عداد سلامة.

```
safetyCounter = 0;
do {
 node = node->Next;
 ...
 safetyCounter++;
 if ( safetyCounter >= SAFETY_LIMIT ) {
 Assert( false ،"Internal Error: Safety-Counter Violation." );
 }
```

هنا شفرة عداد السلامة.

```
  ...
} while ( node->Next != NULL );
```

إن عدادات السلامة ليست علاجًا لجميع الحلقات. تزيد عدادات السلامة المُدخلة في شفرة واحدة مرة واحدة، من التعقيد ويمكن أن تقود لأخطاء إضافية. لأنه لا يتم استخدامها في كل حلقة، فربما تنسى أن تصحح شفرة عداد السلامة عندما تعدل الحلقات في الأجزاء التي تستخدمها من البرنامج. إذا عينت عدادات السلامة كمعيار على مستوى المشروع للحلقات الحرجة، على أي حال، تكون تعلمت أن تتوقعها وعند ذلك لا تعود شفرة عداد السلامة تميل إلى إنتاج أخطاء أكثر من أي شفرة أخرى.

الخروج باكرًا من الحلقة

تزود العديد من اللغات وسائل تسبب إنهاء الحلقة بطريقة ما غير طريقة إتمام شروط حلقة for أو while. في هذا النقاش، عبارة break هي حالة عامة لعبارة break بلغة سي++، وسي# وجافا، ولحلقة Exit-Do وحلقة Exit-For بلغة فيجول بيسك، وللبُنى المشابهة، بما فيها المشبهة بعبارة goto باللغات التي لا تدعم عبارة break مباشرةً. تسبب عبارة break (أو مكافئها) انتهاء الحلقة عبر قناة الخروج الطبيعي، ويقوم البرنامج بمتابعة التنفيذ بدءً من أول عبارة بعد الحلقة.

إن عبارة continue مشابهة لعبارة break بكونها عبارة تحكم إضافية للحلقة. أكثر من كونها تسبب الخروج من الحلقة، على كل حال، تسبب عبارة continue تخطي جسم الحلقة في البرنامج ومتابعة التنفيذ في بداية التكرار التالي للحلقة. عبارة continue هي اختصار لعبارة if-then التي تمنع بقية الحلقة من أن تُنفذ.

خذ بعين الاعتبار استخدام عبارات break أكثر من الأعلام المنطقية في حلقة while. في بعض الحالات، يجعل إضافة إشارات منطقية لحلقة while لمحاكاة الخروج من جسم الحلقة الحلقة صعبة القراءة. يمكنك أحيانًا إزالة عدة مستويات من المسافة البادئة داخل حلقة وتبسيط تحكم الحلقة باستخدام عبارة break فقط بدلًا من سلسلة من اختبارات عبارة if.

يمكن أن يقلل وضع شروط break المتعددة بعبارات منفصلة بجانب الشفرة التي تنتج break، التداخل ويجعل الحلقة أكثر قابلية للقراءة.

كُن حذرًا من حلقة فيها الكثير من عبارات break المبعثرة. يمكن أن توحي الحلقة المتضمنة الكثير من عبارات break بتفكير غير واضح حول بنية الحلقة أو دورها في الشفرة المحيطة. يزيد توالد عبارات break الإمكانية التي تجعل الحلقة أكثر وضوحًا بالتعبير عنها بسلسلة من الحلقات بدلاً من حلقة واحدة بعدة مخارج.

اعتماداً على مقالة "ملاحظة هندسة البرمجيات" (Software Engineering Notes) إن الخطأ البرمجي الذي أسقط أنظمة الهاتف في مدينة نيويورك لمدة 9 ساعات في 15 كانون الثاني 1990، كان بسبب وجود عبارة break زائدة (سين 1990):

مثال بلغة البرمجة سي++ للاستخدام الخاطئ لعبارة break داخل الكتلة do-switch-if.

```
do {
 ...
 switch
 ...
 if () {
  ...
  break;
  ...
 }
 ...
} while ( ... );
```

عبارة breakهذه كانت مُعدة لـ if ولكنها بدلًا من ذلك تخرج من عبارة switch.

لا تشير عبارات break الكثيرة بالضرورة لخطأ، ولكن وجودها في حلقة هو إشارة تحذير، مثل الكناري في منجم فحم يلهث من أجل الهواء بدلاً من الغناء بصوتٍ عالٍ كما يجب أن يكون.

استخدم عبارة continue للاختبارات في أعلى الحلقة. الاستخدام الجيد لعبارة continue هو لتحريك التنفيذ إلى ما بعد جسم الحلقة بعد اختبار شرط في الأعلى. مثلاً، إذا كانت الحلقة تقرأ سجلات، وتهمل سجلات من نوع ما، وتعالج سجلات من نوع آخر، يمكنك وضع اختبار كهذا في أعلى الحلقة:

مثال شفرة زائفة عن استخدام آمن نسبيًا للعبارة continue.

```
while ( not eof( file ) ) do
 read( record ،file )
 if ( record.Type <> targetType ) then
 continue
 -- process record of targetType
 ...
end while
```

يتيح لك استخدام عبارة continue بهذه الطريقة تجنّب اختبار if الذي يبدأ بشكل فعال كامل جسم الحلقة. بالمقابل، إذا وقعت العبارة continue عند منتصف أو نهاية الحلقة، استخدم if بدلًا عنها.

استخدم بنى break الموسومة إذا كانت لغتك تدعمها. تدعم جافا استخدام break الموسومة لمنع نوع من المشاكل تم اختباره في برمجة الهواتف في مدينة نيويورك. يمكن استخدام break الموسومة للخروج من حلقة for، أو عبارة if، أو أي قسم من الشفرة محصورة بقوسين (أرنولد، جوسلينج، وهولمز 2000).

هنا حل ممكن لمشكلة شفرة الهاتف في مدينة نيويورك، حيث تم تغيير لغة البرمجة من سي++ الى جافا لإظهار عبارة break الموسومة.

مثال بلغة البرمجة جافا لاستخدام أفضل لعبارة break الموسومة داخل كتلة do-switch-if.

```
do {
 ...
 switch
 ...
 CALL_CENTER_DOWN:
 if () {
  ...
  break CALL_CENTER_DOWN;
  ...
 }
 ...
} while ( ... );
```

هدف break الموسومة جليّ

استخدم break وcontinue بحذر. ينهي استخدام break إمكانية معاملة الحلقة كصندوق أسود. أن تحدّ نفسك بعبارة واحدة فقط للتحكم بشروط الخروج من حلقة هو طريقة قويّة لتبسيط الحلقات. يجبر استخدام break الشّخص الذي يقرأ الشّفرة أن ينظر إلى داخل الحلقة لفهم كيفية التحكم في الحلقة. وهذا يصعّب فهم الحلقة.

استخدم break فقط عندما تكون قد فكّرت في البدائل. أنت لا تعرف بالضّبط ما إذا كانت break وcontinue بنى جيدة أم سيّئة. يجادل بعض علماء الحاسوب بأنها تقنيّات منطقيّة في البرمجة المنظمة؛ والبعض الآخر يجادلون بأنها ليست كذلك. ولأنك لا تعلم بشكل عام ما إذا كانت continue وbreak صحيحتان أم لا، استخدمهما، لكن فقط مع خوف من أن تكون مخطئًا. إنّه حقا موضوع بسيط: إن لم تستطع الدفاع عن break أو continue، لا تستخدمهما.

التحقق من النّقاط النّهائيّة

لدى الحلقة الواحدة عادةً ثلاث حالات مهمّة: الحالة الأولى، حالة وسطية مختارة بشكل عشوائي، وحالة نهائيّة. عندما تُنشِئ حلقة، راجع الحالات الأولى والوسطيّة والأخيرة عقليًا للتّأكّد من أن الحلقة لا تحوي أية أخطاء "خطأ الخطوة الواحدة". إذا كان لديك حالات خاصّة مختلفة عن الحالة الأولى أو الأخيرة، افحصها أيضًا. إذا كانت الحلقة تحوي حسابات معقّدة، افحصها يدويًا باستخدام الآلة الحاسبة.



إنّ القيام بهذه الفحص هو اختلاف جوهري بين المبرمجين الفعّالين وغير الفعّالين. يقوم المبرمجون الفعّالون بالحسابات الذّهنيّة واليدويّة لأنّهم يعلمون بأنّ هذه القياسات تساعدهم في اكتشاف الأخطاء.

يميل المبرمجون غير الفعّالون للاختبار بعشوائيّة حتى يجدوا خلطة يبدو أنّها تعمل. إذا كانت الحلقة لا تعمل كما يجب، يقوم المبرمج غير الفعّال بتغيير إشارة < إلى إشارة >=. وإذا فشل ذلك، يقوم بتغيير فهرس الحلقة بإضافة أو طرح 1. في النهاية قد يصادف المبرمج الذي يستخدم هذه الطّريقة الخلطة الصّحيحة أو أن يستبدل ببساطة الخطأ الأصلي بخطأ أكثر إتقانًا. وحتّى إذا أنتجت هذه العمليّة العشوائيّة برنامجًا صحيحًا، فإنها لا تنتج لدى المبرمج معرفة سبب صحة البرنامج.

تستطيع توقُّع عدّة فوائد من الحسابات الذّهنيّة واليدويّة. يُنتِج المبدأ العقلي عدداً أقلّ من الأخطاء من خلال كتابة الشفرة الأساسيّة، ويعطي سرعة أكبر في اكتشاف الأخطاء خلال التّصحيح، وفي فهم كلّي أفضل للبرنامج. يعني التّمرين العقلي أنّك تفهم كيف تعمل شفرتك ولست تخمّن فقط.

استخدام متغيّرات الحلقة

ها هنا بعض العناوين لاستخدام متغيّرات الحلقة:

استخدم الأنواع الترتيبية أو التعدادية للحدود في المصفوفات والحلقات[1]. بشكل عام، يجب أن تكون عدّادات الحلقات قيمًا صحيحة. حيث لا تتزايد قيم الفاصلة العائمة بشكل جيّد. على سبيل المثال، يمكن أن تضيف `1.0` إلى `26 742 897.0` وتحصل على `26 742 897.0` عوضًا عن `26 742 898.0`. إذا كانت القيمة المتزايدة هي عدّاد حلقة، سيكون لديك حلقة لا منتهية.



استخدم أسماء متغيّرات ذات معنى لجعل الحلقات المتداخلة قابلة للقراءة. يتم الدخول إلى المصفوفة غالبًا بنفس المتغيّرات التي تستخدم لفهارس الحلقة. إذا كان لديك مصفوفة أحادية البعد، فمن الممكن أن تنجح باستخدام `i`، `j`، `k` للدخول إليها. أما إذا كان لديك مصفوفة ببعدين أو أكثر، يجب

1 **إشارة مرجعية**: لمزيد من التفاصيل حول تسمية متغيرات حلقة انظر "تسمية فهارس حلقة" في القسم 2.11.

عليك استخدام أسماء فهرس ذات معنى لتوضيح ماذا تفعل. توضح أسماء فهرس المصفوفة ذات المعنى كلّاً من هدف الحلقة والجزء من مصفوفة الذي تريد الوصول إليه.

و هنا شفرة لا تُوظِّف هذا المبدأ؛ فهي تستخدم أسماء مبهمة i، j، k بدلًا عن ذلك:



مثال بلغة البرمجة جافا عن أسماء سيئة لمتغير حلقة.

```
for ( int i = 0; i < numPayCodes; i++ ) {
 for ( int j = 0; j < 12; j++ ) {
 for ( int k = 0; k < numDivisions; k++ ) {
  sum = sum + transaction[ j ][ i ][ k ];
 }
 }
}
```

ماذا تعتقد يعني فهرس المصفوفة في transaction؟ هل تخبرك i، j، k أي شيء عن محتويات transaction ؟ إذا كان لديك تصريح ل transaction، هل يمكنك بسهولة تحديد ما إذا ما كان الفهرس في التّرتيب الصحيح؟ ها هي نفس الحلقة مع متغيّرات حلقة أكثر قابليّة للقراءة:

مثال بلغة البرمجة جافا عن أسماء جيدة لمتغير الحلقة.

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {
 for (int month = 0; month < 12; month++ ) {
 for ( int divisionIdx = 0; divisionIdx < numDivisions; divisionIdx++ )
{
  sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];
 }
 }
}
```

ماذا تعتقد تعني فهارس المصفوفة في transaction هذه المرّة؟ في هذه الحالة، الإجابة أسهل لأن أسماء المتغيّرات payCodeIdx، month، وdivisionIdx تخبرك أكثر بكثير من i، j، k. يستطيع الحاسوب قراءة نسختي الحلقة بسهولة متعادلة. ويستطيع الناس قراءة النّسخة الثّانية بشكل أكثر سهولة من الأولى، ولكن تعتبر النّسخة الثانية أفضل لأن جمهورك الأساسي هو من الناس وليس الحواسيب.

استخدم أسماء ذات معنى لتتجنّب اختلاط الكلام لفهارس الحلقة. يمكن أن يُصعّد التعوُّد على استخدام i ، j، k أخطاء اختلاط الفهارس- استخدام نفس اسم الفهرس لغرضين مختلفين.

انظر إلى هذا المثال:

مثال بلغة البرمجة سي++ عن خطأ تبديل الفهرس.

```
for ( i = 0; i < numPayCodes; i++ ) {
```

استخدمنا i هنا أول مرة...

```
// lots of code
...
for ( j = 0; j < 12; j++ ) {
// lots of code
...
for ( i = 0; i < numDivisions; i++ ) {
 sum = sum + transaction[ j ][ i ][ k ];
}
}
}
```

...وهنا مرة ثانية.

أصبح استخدام i عادة لدرجة أنّها تُستخدم مرّتين في نفس البنية المتداخلة. تتداخل حلقة for الثّانية التي تتحكّم فيها i مع الأولى، وهذا هو اختلاط الفهارس. كان سيمنع المشكلة استخدام أسماء أكثر معنى من i، j، k. بشكل عام، إذا كان هيكل الحلقة يحوي أكثر من سطرين، أو إذا كان من الممكن أن تكبر، أو إذا كانت في مجموعة من الحلقات المتداخلة، تجنّب i، j، k.

احصر نطاق متغيّرات فهرسة الحلقة إلى الحلقة نفسها فقط. إن اختلاط فهارس الحلقة واستخدامها خارج حلقاتها هو مشكلة كبيرة لدرجة أن مصمّمي لغة آدا (Ada) قرّروا ابتكار فهرس لحلقة for غير صالحة للاستخدام خارج حلقته؛ حيث تولد محاولة استخدام واحدة خارج حلقة for الخاصة بها خطأً في وقت الترجمة.

طبّقت لغة البرمجة سي++ وجافا نفس الفكرة إلى حد ما – حيث إنهم يسمحون باستخدام فهرس الحلقة بداخل حلقة أخرى، لكنهم لا يطلبونها. في المثال في الصفحة 542، متغيّر recordCount يمكن التّصريح عنه داخل عبارة for، ممّا سيحدّ تأثيره إلى حلقة for فقط، مثال هذا:

مثال بلغة البرمجة سي++ عن التصريح عن متغير فهرس حلقة داخل حلقة for.

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {

 // recordCount شفرة الحلقة التي تستخدم
}
```

من ناحية المبدأ، يمكن أن تسمح هذه التّقنيّة بإعادة التّصريح باستخدام recordCount في حلقات عدّة بدون خطر سوء استعمال recordCount مختلفتين. هذا الاستخدام يعطي شفرة تبدو كهذه:

مثال بلغة البرمجة سي++ عن التصريح عن فهارس الحلقة داخل الحلقات وإعادة استخدامهم بشكل آمن- ربما.

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
 // recordCount شفرة الحلقة التي تستخدم
}
// شفرة التعشيش أو التداخل
```

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
 // recordCount شفرة حلقة إضافية تستخدم
}
```

تساعد هذه التّقنيّة على توثيق الهدف من متغيّر recordCount، لكن لا تعتمد على المترجم البرمجي لإلزامه بمجال recordCount. يقول القسم 1.3.3.6 من كتاب (لغة سي++ للبرمجة، ستروسترب 1997) أن recordCount يجب أن يكون له مجال محصور بحلقته. عندما تحقّقت من هذا الأمر بشكل عملي مع ثلاث مترجمات سي++، حصلت على ثلاث نتائج مختلفة:

- المترجم الأول علّمَ recordCount في حلقة for الثّانية للتّصريح عن عدة متغيّرات وأعطى خطأً.
- المترجم الثّاني قَبِلَ recordCount في حلقة for الثّانية لكن سمح باستخدامها خارج حلقة for الأولى.
- المترجم الثّالث سمح بالاستخدامين ل recordCount ولم يسمح لأيّ منهما أن تستخدم خارج حلقة for حيث تمّ التّصريح عنها.

بما أن هذه هي الحال مع أغلبيّة ميّزات اللّغة المخفيّة، فتطبيقات المترجم يمكن أن تتنوّع.

كم يجب أن يكون طول الحلقة؟

يمكن أن يقاس طول الحلقة بعدد الأسطر أو بعمق التّداخل. وهنا بعض التّوجيهات:

اجعل حلقاتك قصيرة بما فيه الكفاية لتظهر كل شيء دفعة واحدة. إذا كنت تنظر عادةً إلى الحلقات على شاشتك وشاشتك تعرض 50 سطرًا، هذا يفرض عليك حدًا من 50 سطر. اقترح الخبراء أن يكون حدّ طول الحلقة هو صفحة واحدة. عندما تبدأ بتقدير مبدأ كتابة شفرة بسيطة، نادرًا ما ستكتب حلقات أطول من 15 أو 20 سطرًا.

حدّد التّداخل بثلاث مراحل[1]. أظهرت الدّراسات أن قدرة المبرمجين على فهم الحلقة يتناقص بشكل ملحوظ بعد ثلاث مراحل من التّداخل (يوردون 1986). إذا كنت ذاهبًا بعد ذلك الرّقم من المراحل، اجعل الحلقة أقصر وذلك عبر كسر جزء منها ووضعه داخل إجرائيّة أو تبسيط بنية التحكّم.

نقل ما بداخل الحلقة إلى إجرائية في الحلقات الطّويلة. إذا كانت الحلقة مصمّمة بشكل جيّد، غالبًا يمكن نقل الشفرة بداخل الحلقة إلى إجرائيّة أو أكثر ومن ثمّ استدعائها من داخل الحلقة.

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن تبسيط التّداخل راجع القسم 4.19 "ترويض التّداخل العميق بشكل خطر "

اجعل الحلقات الطّويلة واضحة بشكل خاص. يضيف الطّول التّعقيد. إذا كتبت حلقة قصيرة، يمكنك استخدام بنى تحكّم أكثر خطورة مثل break وcontinue، وعدّة مخارج، وشروط الإنهاء المعقّدة، وهكذا. إذا كتبت حلقة أطول وشعرت بأي اهتمام نحو قارئك، ستعطي الحلقة خروجًا واحدًا وستجعل شروط الخروج واضحة بشكل غير قابل للخطأ.

3.16 إنشاء الحلقات بسهولة – من الداخل للخارج

إذا كان لديك أحيانًا مشاكل في كتابة شفرة حلقة معقّدة – الأمر الذي يعاني منه معظم المبرمجين– تستطيع استخدام تقنيّة سهلة لتكتبها بشكل صحيح من المرّة الأولى. وها هي العمليّة العامّة. ابدأ بحالة واحدة، اكتب شفرة تلك الحالة بالاستعانة بالأحرف، قم بإزاحتها إلى اليمين قليلًا، ثم ضع حلقة حولها، ثم استبدل الأحرف بفهارس الحلقة أو التّعابير المحسوبة. ثم ضع حلقة أخرى حول ذلك، بحسب الضّرورة، ثمّ استبدل المزيد من الأحرف. استمرّ بالعمليّة بالقدر الذي تحتاجه. عندما تنتهي، أضف كل التهيئة الضّروريّة. بما أنّك بدأت بحالة بسيطة وعملت على تعميمها، يمكن أن تسمّيها كتابة الشفرة من الدّاخل إلى الخارج.

افترض أنّك تكتب برنامجًا لشركة تأمين[1]. لديها معدّلات تأمين مدى الحياة تتنوّع بحسب عمر وجنس الشّخص. عملك هنا هو أن تكتب إجرائيّة تحسب مجموع أقساط التّأمين مدى الحياة لمجموعة. تحتاج حلقة تأخذ المعدّل لكل شخص في القائمة وتضيفه إلى المجموع. وهنا كيف ستفعلها.

أوّلًا، في التّعليقات على الموضوع، اكتب الخطوات التي يحتاج هيكل الحلقة فعلها. من الأسهل كتابة ما تريد فعله عندما لا تفكّر بتفاصيل النّحو، فهرس الحلقة، فهرسات المصفوفة، وهكذا.

الخطوة 1: إنشاء حلقة من الداخل الى الخارج (مثال شفرة زائفة).

-- الحصول على المعدل من الجدول.

-- إضافة المعدل إلى المجموع.

```
الخطوة 1: إنشاء حلقة من الداخل إلى الخارج
(مثال شفرة زائفة)
-- get rate from table
-- add rate to total
```

[1] **إشارة مرجعية:** تشفير حلقة من الدّاخل للخارج مشابه لعمليّة وُصفَت في الفصل التّاسع "عمليّة برمجة السودوكود".

ثانيًا، حوّل التعليقات في هيكل الحلقة إلى شفرة، بقدر ما يمكنك، بدون أن تكتب كامل الشّفرة. في هذه الحالة، خذ معدّل شخص واحد وأضفه إلى المجموع الكلّي. استخدم بيانات صلبة محدّدة وليس ملخّصات.

الخطوة 2: إنشاء حلقة من الداخل إلى الخارج (مثال شفرة زائفة).

```
rate = table[ ]
totalRate = totalRate + rate
```

ليس لدى tableأيّة فهارس بعد.

يفترض المثال أن table هو مصفوفة تحمل بيانات المعدّل. ليس عليك القلق حول فهارس المصفوفة مبدئيًا. rate هو متغيّر يحمل بيانات المعدّل المختارة من جدول المعدّلات. وكذلك، totalRate هو متغيّر يحمل مجموع المعدّلات.

بعد ذلك، ضع الفهارس في مصفوفة table:

الخطوة 3: إنشاء حلقة من الداخل إلى الخارج (مثال الشفرة الزائفة).

```
rate = table[ census.Age ][ census.Gender ]
totalRate = totalRate + rate
```

يتم الولوج إلى هذه المصفوفة عبر العمر والجنس، لذلك تُستخدم census.Age وcensus.Gender كفهارس في المصفوفة. يفترض المثال أن census هي بنية تحمل معلومات عن الأشخاص في المجموعة التي سيتم فحص معدّلاتها.

الخطوة التّالية هي أن تبني حلقة حول العبارات الموجودة. بما أنّه من المفترض للحلقة أن تقوم بحساب المعدّلات لكل شخص في المجموعة، يجب أن يُدخل كل شخص إلى الحلقة.

الخطوة 4: إنشاء شفرة من الداخل إلى الخارج (مثال الشفرة الزائفة).

```
For person = firstPerson to lastPerson
 rate = table[ census.Age ،census.Gender ]
 totalRate = totalRate + rate
End For
```

كل ما عليك فعله هنا هو أن تضع حلقة for حول الشّفرة الموجودة ثم قم بإزاحة للشّفرة الموجودة مسبقًا وضعها داخل ثنائيّة begin-end. أخيرًا، تحقّق من أنّ المتغيّرات التي تعتمد على فهرس حلقة person قد تم تعميمها. في هذه الحالة، متغيّر census يتغيّر مع person، ولذلك يجب تعميمه بشكل صحيح.

أخيرًا، اكتب أي تهيئة تحتاجها. في هذه الحالة، يجب أن تتم تهيئته متغيّر totalRate.

الخطوة الأخيرة: إنشاء حلقة من الداخل إلى الخارج (مثال الشفرة الزائفة).

```
totalRate = 0
For person = firstPerson to lastPerson
 rate = table[ census[ person ].Age ،census[ person ].Gender ]
 totalRate = totalRate + rate
End For
```

إذا كان عليك وضع حلقة أخرى حول حلقة person، أكمل بنفس الطّريقة. ليس عليك أن تتبع الخطوات بدقّة. الفكرة هي أن تبدأ بشيء صلب، وأن تفكّر بشيء واحد كل مرّة، وأن تبني الحلقة من مكوّنات بسيطة. خذ خطوات بسيطة ومفهومة وأنت تجعل الحلقة أكثر عموميَّة وتعقيدًا. بهذه الطّريقة، فأنت تقلّل كميّة الشّفرة التي عليك أن تركّز عليها في كل مرّة وبالتّالي تقلّل فرصة الخطأ.

4.16 التراسل بين الحلقات والمصفوفات

الحلقات والمصفوفات غالبًا ما تكون مترابطة[1]. في كثير من الحالات، تنشأ الحلقة لتقوم بالتّلاعب بالمصفوفة، وعدّادات الحلقة تتجاوب مع فهارس المصفوفة واحداً لواحد. على سبيل المثال، تتجاوب فهارس حلقة for هذه في لغة جافا مع فهارس المصفوفة:

مثال بلغة البرمجة جافا عن ضرب المصفوفات.

```
for ( int row = 0; row < maxRows; row++ ) {
 for ( int column = 0; column < maxCols; column++ ) {
 product[ row ][ column ] = a[ row ][ column ] * b[ row ][ column ];
 }
}
```

في جافا، الحلقة ضروريّة لعمليّة المصفوفة هذه. لكن من الجيد ملاحظة أن بنى الحلقات والمصفوفات ليستا مرتبطتين أصلًا. بعض اللّغات، وبالأخصّ APL و90 Fortran وما بعدها، تزوّد بعمليات مصفوفات قويّة تلغي الحاجة للحلقة كالمثال السّابق. وهنا شفرة APL تنفّذ نفس العمليّة:

مثال APL لضرب مصفوفة.

```
product <- a x b
```

1 **إشارة مرجعية**: لنقاش أكثر حول التّجاوب بين الحلقات والمصفوفات، راجع القسم 7.10 "العلاقة بين نوع البيانات وبنى التّحكّم ".

لغة APL أبسط وأقلّ عرضة للخطأ. فهي تستخدم ثلاث معاملات فقط، وجزء من جافا يستخدم 17. ليس لديها متغيّرات حلقة، أو فهرس مصفوفة، أو بنى تحكّم ليتمّ تشفيرها بشكل خاطئ.

نقطة معيّنة من هذا المثال هي أنّك عندما تقوم ببعض البرمجة لحلّ المشكلة والبعض الآخر لحلّها بلغة معيّنة. تؤثّر اللّغة التي تختارها لحلّ مشكلة ما على حلّك بشكل كبير جدًّا.

لائحة اختبار: الحلقات1

اختيار الحلقة وإنشائها

- هل تستخدم حلقة while بدلًا عن حلقة for إذا كانت مناسبة؟
- هل صُمّمَت الحلقة من الدّاخل إلى الخارج؟

الدخول إلى الحلقة

- هل تمّ الدخول إلى الحلقة من الأعلى؟
- هل شفرة التّهيئة موجودة مباشرةً قبل الحلقة؟
- إذا كانت الحلقة لانهائيّة أو حلقة حدث، هل تم إنشاؤها بنظافة عوضًا عن استخدام حل متسرّع للمشكلة مثل for i =1 to 9999؟
- إذا كانت حلقة for لجافا أو سي++ أو سي هل رأس الحلقة محجوز فقط لشفرة التحكّم بالحلقة؟

داخل الحلقة

- هل تستخدم الحلقة { و} أو ما يعادلها لتغليف هيكل الحلقة ومنع المشاكل التي تنشأ من التّعديلات غير الصّحيحة؟
- هل هيكل الحلقة يحوي شيئاً بداخله؟ هل هو غير فارغ؟
- هل التّدابير التّحضيريّة مجتمعة؛ إما في بداية أو نهاية الحلقة؟
- هل تقوم الحلقة بوظيفة واحدة وواحدة فقط، كما تفعل إجرائيّة معرّفة بشكل جيّد؟
- هل الحلقة قصيرة بما فيه الكفاية لإظهار كل شيء دفعة واحدة؟
- هل الحلقة متداخلة بثلاث مراحل أو أقلّ؟
- هل تم تحريك محتويات الحلقة إلى الإجرائيات الخاصة بها؟
- إذا كانت الحلقة طويلة، هل هي واضحة بشكل خاصّ؟

فهرس الحلقة

[1] cc2e.com/1616

- إذا كانت الحلقة حلقة for، هل تقوم الشّفرة بداخلها بتجنّب التّداخل مع فهرس الحلقة؟
- هل يُســتخدَم متغيّر لتخزين قيم فهرس الحلقة المهمّة عوضًــا عن اســتخدام فهرس الحلقة خارج الحلقة؟
- هل فهرس الحلقة من النّوع الترتيبي أو من النّوع التعدادي؟
- هل لفهرس الحلقة أسماء ذات معنى؟
- هل تتجنّب الحلقة مشكلة تداخل الفهارس؟

الخروج من الحلقة

- هل تقوم الحلقة بالإنهاء تحت كل الظّروف الممكنة؟
- هل تستخدم الحلقة عدّادات السّلامة- إذا تم تقرير عدّادات السلامة كمعيار؟
- هل شرط إنهاء الحلقة واضح؟
- إذا تمّ استخدام break أو continue، هل هما صحيحتان؟

نقاط مفتاحية

- الحلقات معقّدة. يُساعد الإبقاء على بساطتها قرّاء شفرتك.
- تتضـمّن التّقنيّات الحفاظ على بسـاطة الحلقات تَجنّب الأنواع الغريبة من الحلقات، وتقليص التّداخل، وتوضيح الدّخول والخروج، والإبقاء على شفرة التّدابير التّحضيريّة في مكان واحد.
- قد يكون فهرس الحلقة مصدرًا كبيرًا للإزعاج. سمّهم بوضوح، واستخدمهم لهدف واحد فقط.
- فكّر بالحلقة بحذر للتأكّد من أنّها تعمل بشكل طبيعي عند كل حالة وتنتهي تحت كلّ الظّروف الممكنة.

الفصل 17

بنى التّحكم غير العادية

المحتويات[1]

مواضيع ذات صلة

- قضايا التحكم العامة: الفصل 19
- الشفرة الخطيّة: الفصل 14
- استخدام الشّرطيات: الفصل 15
- استخدام الحلقات: الفصل 16
- معالجة الاستثناءات: القسم 8-4

تقع العديد من بنى التحكم في منطقة مظلمة ضبابيّة، بين أن تكون في المقدّمة أو أن تكون غير موثوقة وغير مثبتة– وغالبًا تقع في المكانين معًا في نفس الوقت! هذه البنى غير متوفرة في كل اللغات لكن يمكن أن تكون مفيدة عندما تستخدم بعناية في اللّغات التي تقدمها.

17. 1 إعادات متعددة من إجرائيّة ما

تدعم معظم لغات البرمجة بعض الوسائل للخروج من الإجرائيّة من منتصفها خلال تنفيذ الإجرائية. إن عبارات "الإعادة"(return) و"الخروج"(exit) هي بنى تحكم، تسمح للبرامج بالخروج من الإجرائيّة حالًا. حيث أنها تسمح للإجرائيّة أن تنهي نفسها من خلال قناة الخروج العادية، معيدةً التّحكم إلى الإجرائيّة المستدعية.

[1] cc2e.com/1778

تُستخدم كلمة "return" هنا كمصطلح عام للكلمة المفتاحية *return* في لغات البرمجة سي++ وجافا، والكلمات المفتاحية (exit sub) و(exit function) في "مايكروسوفت فيجوال بيسيك "، وبنى أخرى مشابهة. وهنا بعض التوجيهات لاستخدام العبارة "return":



استخدم عبارة "return" عندما تُحسّن القدرة على القراءة. في إجرائيّة معيّنة، عندما تعرف الإجابة، فإنك تريد العودة إلى الإجرائيّة المستدعية في الحال. إذا كانت الإجرائيّة معرفة بحيث لا يتطلب أي تنظيف عند اكتشاف خطأ ما، فإنه يعني عدم الرجوع الفوري أنك بحاجة لكتابة المزيد من الشفرة.

فيما يلي مثال جيّد لحالة يكون فيها الإعادة من أماكن متعددة في الإجرائيّة منطقيًا:

مثال في سي++ يوضح الإعادات المتعددة الجيّدة من إجرائيّة:

```
Comparison Compare( int value1, int value2 ) {
    if ( value1 < value2 ) {
        return Comparison_LessThan;
    }
    else if ( value1 > value2 ) {
        return Comparison_GreaterThan;
    }
    return Comparison_Equal;
}
```

هذه الإجرائيّة تعيد النوع التعدادي "*comparison*"

إن الأمثلة الأخرى أقلّ وضوحًا، كما يوضّح القسم الفرعي التّالي.

استخدم جمل الحماية (الإعادة أو الخروج المبكر) لتسهيل معالجة الأخطاء المعقدة. يمكن أن تؤدي الشفرة التي يكون عليها التحقّق من عدد كبير من الشروط قبل القيام بأعمالها الاعتبارية إلى شفرة مجزأة بشكل كبير ويمكن أن تحجب العمليّة الأساسيّة، كما هو موضح هنا:

شفرة فيجوال بيسك تحجب الحالة الأساسيّة:

```
If file.validName() Then
    If file.Open() Then
        If encryptionKey.valid() Then
            If file.Decrypt( encryptionKey ) Then
                ' lots of code
                ...
            End If
        End If
    End If
End If
```

← هذه هي الشفرة للحالة الأساسيّة.

تضمين الجسم الرئيسي للإجرائيّة بداخل أربع عبارات "If" بشع من الناحية الجمالية، وخصوصًا إذا كان هناك الكثير من الشفرة في معظم القسم الداخلي لعبارة "if". في هذه الحالات، دفق الشفرة يكون أوضح إذا تفقدت الحالات الخاطئة أولًا، منظفًا الطّريق للمسار المعتبر خلال الشفرة. وهنا كيف يمكن أن يبدو ذلك:

شفرة فيجوال بيسك بسيطة تستخدم جمل الحماية لتوضيح الحالة الاعتبارية:

```
' set up, bailing out if errors are found
If Not file.validName() Then Exit Sub
If Not file.Open() Then Exit Sub
If Not encryptionKey.valid() Then Exit Sub
If Not file.Decrypt( encryptionKey ) Then Exit Sub
' lots of code
...
```

الشّفرة البسيطة تجعل هذه الطّريقة تبدو كحلّ مرتّب. لكن الشفرة المنتجة تتطلب تدابيرًا تحضيريّة أكثر شموليّة أو حذف للبيانات عند اكتشاف شرط خاطئ. وهذا مثال أكثر واقعية:

شفرة فيجوال بيسيك أكثر واقعية تستخدم جمل الحماية لتوضيح الحالة الاعتباريّة:

```
' set up, bailing out if errors are found
If Not file.validName() Then
    errorStatus = FileError_InvalidFileName
    Exit Sub
End If

If Not file.Open() Then
    errorStatus = FileError_CantOpenFile
    Exit Sub
End If

If Not encryptionKey.valid() Then
    errorStatus = FileError_InvalidEncryptionKey
    Exit Sub
End If

If Not file.Decrypt( encryptionKey ) Then
    errorStatus = FileError_CantDecryptFile
    Exit Sub
End If

' lots of code    <-- هذه هي الشفرة للحالة الأساسيّة.
...
```

مع شفرة بحجم شفرة البرامج المنتجة، يُنتِج أسلوب "Exit Sub" كمية ملحوظة من الشّفرة قبل تنفيذ العمليّة الأساسيّة. لكن يتجنّب أسلوب "Exit Sub" التّداخل الكبير كما في المثال الأوّل، وإذا كانت الشّفرة مُدّدَت لتعرض خيار متحوّل "errorStatus"، فيقوم أسلوب "Sub" Exit بعمل أفضل في إبقاء الجمل مترابطة مع بعضها. وعندما ينتهي العمل، يبدو أسلوب "Exit Sub" أكثر قابلية للقراءة ومدعومًا أكثر، ولكن ليس بشكل مبالغ به.

قلّل عدد الإعادات في كل إجرائيّة. إنّه لمن الصّعب فهم إجرائيّة معيّنة عندما تقرأها من الأسفل، وأنت غير مدرك لاحتماليّة أنها استخدمت "return" في مكان ما في أعلى الشفرة. ولهذا السّبب، استخدم "return" بتعقُّل – فقط عندما تُحسّن القراءة.



17. 2 العوديّة

في العوديّة، تحل الإجرائيّة قسم صغير من المشكلة نفسها، حيث تقسم المشكلة إلى أجزاء أصغر، ثم تستدعي نفسها لحلّ كلّ من الأجزاء الصّغيرة. تُستخدم العوديّة عندما تكون الأجزاء الصغيرة للمشكلة قابلة للحل بسهولة، والقسم الكبير منها قابل للتّفكيك إلى أجزاء أصغر بسهولة.

العوديّة غالبًا ليست مفيدة، لكن عند استخدامها بتعقُّل فهي تُنتج حلولًا أنيقة، كما في هذا المثال الّذي تكون فيه خوارزميّة الترتيب هي استخدام ممتاز للعوديّة:

مثال بلغة جافا لحل خوارزمية الترتيب باستخدام العوديّة:

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {
    if ( lastIndex > firstIndex ) {
        int midPoint = Partition( firstIndex, lastIndex, names );
      QuickSort( firstIndex, midPoint-1, names );
        QuickSort( midPoint+1, lastIndex, names );
    }
}
```

هنا نقوم باستدعاء العوديّة

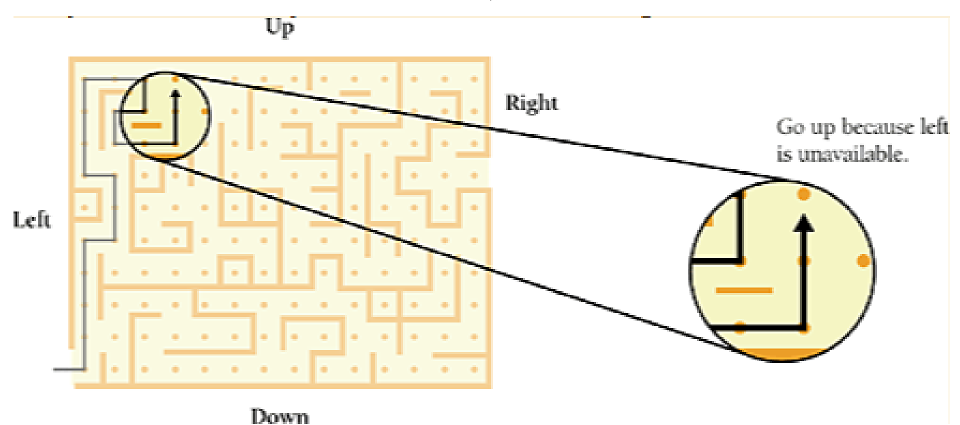
في هذه الحالة، تقسم خوارزميّة الترتيب المصفوفة إلى اثنتين ثم تستدعي نفسها لترتب كل نصف من المصفوفة. عندما تستدعي نفسها لمصفوفة فرعية أصغر من أن يتم ترتيبها – مثل (firstIndex => lastIndex) – فهي تتوقف عن استدعاء نفسها.

من أجل مجموعة صغيرة من المشاكل، يمكن أن تُنتِج العوديّة حلولًا بسيطة وأنيقة. من أجل مجموعة أكبر بقليل من المشاكل، يمكن أن تُنتِج حلولًا بسيطة وأنيقة – وصعبة الفهم. من أجل معظم المشاكل، فهي تُنتج حلولًا معقدة بشكل هائل – في هذه الحالات، التكرار البسيط عادة ما يكون أكثر قابلية للفهم. لذلك استخدم العوديّة بشكل انتقائي.

أمثلة عن العوديّة

افترض أن لديك نوع من البيانات، الّتي تمثل متاهة. المتاهة هي شبكة بشكل أساسي، وعند كل نقطة على الشّبكة تكون قادرً على الالتفاف يسارًا، أو يمينًا، تتحرك للأعلى، أو للأسفل. وغالبًا ستكون قادرًا على التحرك في أكثر من اتّجاه.

كيف تكتب برنامجًا ليجد طريقه خلال المتاهة، كما هو موضح في الشّكل 17-1؟ إذا استخدمت العوديّة فإن الجواب واضح. تبدأ في بداية المتاهة ثمّ تجرب كل الطرق الممكنة حتى تجد طريقك لخارج المتاهة. في أول مرة تزور فيها نقطة ما، ستحاول التحرك لليسار، إذا لم تستطع التحرك يسارًا، ستحاول التحرك لأعلى وأسفل، وإذا لم تستطع التحرك لأعلى وأسفل، ستحاول التحرّك يمينًا. ليس عليك القلق من أن تضيع لأنّك ترمي بضع فتات من الخبز عند كل نقطة تزورها، وبالتالي لا يمكن أن تزور نفس المكان مرتين.



الشكل 17-1: العوديّة يمكن أن تكون أداة قيّمة في الحرب ضدّ التعقيد – عند استخدامها ضدّ مشاكل ملائمة

شفرة العوديّة تبدو كهذه:

مثال سي ++ للتحرّك ضمن متاهة بشكل عوديّ:

```
   bool FindPathThroughMaze( Maze maze, Point position ) {
       // if the position has already been tried, don't try it
again
       if ( AlreadyTried( maze, position ) ) {
           return false;
       }

       // if this position is the exit, declare success
       if ( ThisIsTheExit( maze, position ) ) {
           return true;
       }

       // remember that this position has been tried
       RememberPosition( maze, position );

       // check the paths to the left, up, down, and to the
right; if
       // any path is successful, stop looking
       if ( MoveLeft( maze, position, &newPosition ) ) {
           if ( FindPathThroughMaze( maze, newPosition ) ) {
               return true;
           }
       }

       if ( MoveUp( maze, position, &newPosition ) ) {
           if ( FindPathThroughMaze( maze, newPosition ) ) {
               return true;
           }
       }

       if ( MoveDown( maze, position, &newPosition ) ) {
           if ( FindPathThroughMaze( maze, newPosition ) ) {
               return true;
           }
       }

       if ( MoveRight( maze, position, &newPosition ) ) {
           if ( FindPathThroughMaze( maze, newPosition ) ) {
               return true;
           }
       }
       return false;
   }
```

يتحقّق أوّل سطر من الشّفرة ليرى إذا كان الموضع قد تمّ تجريبه أم لا. مفتاح مهم في كتابة إجرائيّة عوديّة وهو منع حصول عوديّة لانهائية. وفي هذه الحالة، إذا لم تتحقّق إذا كنت قد جربت النّقطة، قد تبقى تجرّبها بشكل لانهائي.

تتحقق الجملة الثّانية إذا كان هذا الموضع هو المخرج من المتاهة. إذا كان كذلك، وكان – "ThisIsTheExit" يعيد "true"، فالإجرائيّة بأكملها تعيد "true".

تتذكّر الجملة الثّالثة ما إذا كان الموضع قد تمّت زيارته أم لا. وهذا يمنع العوديّة اللّانهائية الّذي قد ينتج عن مسار دائري.

تحاول الأسطر الباقية في الإجرائيّة البحث عن طريق إلى اليسار، وإلى الأعلى، وإلى الأسفل، وإلى اليمين. توقف الشفرة العوديّة إذا أعادت الإجرائيّة "true"، وهذا يحصل عندما تجد الإجرائيّة طريقًا في المتاهة.

المنطق المستخدم في هذا الإجرائيّة واضح جدًا. يشعر معظم النّاس ببعض من عدم الرّاحة مبدئيًا عند استخدام العوديّة لأنّها ذاتيّة المرجع. في هذه الحالة، أي حل بديل قد يكون أكثر تعقيدًا بكثير والعوديّة تعمل بشكل جيّد.

نصائح في استخدام العوديّة

أبقِ هذه النصائح في عقلك أثناء استخدام العوديّة:

احرص على أن تتوقّف العوديّة. تفقّد الإجرائيّة للحرص أنها تتضمّن مسارًا ليس عوديًا. يعني هذا في العادة أن الإجرائيّة لديها اختبار يوقف العوديّة عندما لا يكون هناك حاجة لها. في مثال المتاهة، تضمن الاختبارات لـ"AlreadyTried()" (حاولت هذا المكان مسبقًا) و"ThisIsTheExit()" (هذا هو المخرج) توقف العوديّة.

استخدم عدّادات السّلامة لمنع العوديّة اللّانهائية. إذا كنت تستخدم العوديّة في حالة لا تسمح باختبار بسيط مثل الذي استخدم أعلاه، استخدم عدّاد السّلامة لمنع العوديّة اللّانهائية. يجب أن يكون عدّاد السلامة متحوّلًا لا يتمّ إعادة إنشائه في كل مرّة تستدعي الإجرائيّة. استخدم متحوّل صفّ أو مرّر عداد السّلامة كوسيط. وهذا مثال على ذلك:

مثال فيجوال بيسيك لاستخدام عدّاد السّلامة لمنع العوديّة اللانهائية

```
Public Sub RecursiveProc( ByRef safetyCounter As Integer )
    If ( safetyCounter > SAFETY_LIMIT ) Then
        Exit Sub
    End If
    safetyCounter = safetyCounter + 1
    ...
    RecursiveProc( safetyCounter )
End Sub
```

الإجرائيّة العوديّة يجب أن تكون قادرة على تغيير قيمة "SafetyCounter"، لذلك في الفيجوال بيسيك تكون وسيط بالمرجع "ByRef".

في هذه الحالة إذا كانت الإجرائيّة تتجاوز حدّ السّلامة، فإنها توقف العوديّة.

إذا لم ترغب باستخدام عدّاد السّلامة كوسيط بشكل صريح، يمكنك استخدام متغيّر عضو في سي++، أو جافا، أو فيجوال بيسيك، أو ما يعادله في لغات أخرى.

قيّد العوديّة في إجرائيّة واحدة. العوديّة الحلقيّة ("أ" يستدعي "ب" يستدعي "ج" يستدعي "أ") خطيرة لأنّه من الصّعب اكتشافها. إدارة العوديّة بشكل عقلي في إجرائية واحدة صعب بما فيه الكفاية، فهم العوديّة التي تمتد عبر الإجرائيات أمرٌ أصعب. إذا كان لديك عوديّة حلقيّة، يمكنك عادةً إعادة تصميم الإجرائيّة بحيث

تكون العوديّة محصورة في إجرائية واحدة. إذا لم تستطع ولم تزل تعتقد أن العوديّة هي أفضل حل ممكن، استخدم عدادات السّلامة كبوليصة تأمين للعوديّة.

انتبه على المكدّس. مع العوديّة، ليس لديك أيّ ضمانات عن كمية مساحة المكدّس التي يستخدمها برنامجك ومن الصّعب أن تتوقُّع مسبقًا كيف سيتصرّف أثناء تشغيله، على أيّة حال، يمكنك أن تأخذ عدّة خطوات لتتحكم في تصرّفه أثناء وقت التّشغيل.

أولًا إذا استخدمت عدّاد سلامة، أحد الاعتبارات هو وضع حدّ لكميّة المكدّس التي تريد أن تخصّصها في الإجرائية العوديّة. عيّن حدّ السّلامة ليكون منخفضاً بما يكفي لمنع طفْح المكدّس.

ثانيًا، راقب مخصّصات المتغيّرات المحليّة في عمليات العوديّة، وخصوصًا الأشياء المتعلّقة بالذّاكرة المركّزة. بمعنى آخر، استخدم "new" لإنشاء أغراض متراكمة عوضًا عن أن تدع المترجم ينشئ الأغراض تلقائيًّا على المكدّس.

لا تستخدم العوديّة للعامليّ(!) (factorials) أو لأعداد فيبوناتشي: مشكلة كتب علم الحاسوب أنها تقدّم أمثلة سخيفة للعوديّة. الأمثلة المعتادة هي حساب عامليّ (!) أو حساب متتالية فيبوناتشي. العوديّة أداة قويّة، ومن السّخافة استخدامها في أيّ من الحالتين السّابقتين.

إذا قام مبرمج يعمل لديّ باستخدام العوديّة لحساب عامليّ، كنت سأستأجر مبرمجًا آخر. وهذه نسخة العوديّة لإجرائية عامليّ:



مثال جافا لحلّ غير مُناسب: استخدام العوديّة لحساب عامليّ

```
int Factorial( int number ) {
    if ( number == 1 ) {
        return 1;
    }
    else {
        return number * Factorial( number - 1 );
    }
}
```

بالإضافة لكونه بطيئًا وجعله لاستخدام ذاكرة وقت التشغيل غير متوقّعًا، نسخة العوديّة لهذه الاجرائيّة أصعب للفهم من النّسخة التكرارية. فيكون لدينا:

مثال جافا لحل مُناسب: استخدام التكرار لحساب دالّة عامليّ:

```
int Factorial( int number ) {
    int intermediateResult = 1;
    for ( int factor = 2; factor <= number; factor++ ) {
        intermediateResult = intermediateResult * factor;
    }
    return intermediateResult;
}
```

يمكنك استخلاص ثلاثة دروس من هذا المثال. أولًا، كتب علم الحاسوب لا تخدم العالم أبدًا بهذه الأمثلة عن العوديّة. ثانيًا، والأهمّ، العوديّة أداة أقوى بكثير من أن تستخدم بشكل غير واضح في حساب العامليّ أو أعداد فيبوناتشي. ثالثًا، والأكثر أهمية، يجب أن تأخذ بعين الاعتبار استخدام بدائل للعوديّة قبل العمل بها. تستطيع أن تفعل كل ما تفعله بالعوديّة باستخدام المكدّس أو التكرارات. في بعض الأحيان أحد المنهجيات تعمل بشكل أفضل من البقيّة، وأحيانا البعض الآخر يعمل بشكل أفضل. خذ الاثنتين بعين الاعتبار قبل أن تختار أي واحدة.

17. 3 عبارة goto (الذهاب إلى)

لربّما تعتقد أن الجدال حول "goto" قد انقرض، لكن جولة سريعة حول مستودعات شفرة المصدر مثل "sourceforge.net" تظهر أنّ "goto" مازالت حيّة وتعيش في أعماق مخدّمات شركتك. بالإضافة، المكافئات الحديثة لجدال "goto" ما زالت تظهر في قضايا مختلفة، متضمّنةً الجدالات حول الإعادات المتعدّدة، والمخارج المتعددة من الحلقات، والمخارج المسمّاة من الحلقات، ومعالجة الأخطاء، ومعالجة الاستثناءات.

الجدال ضد "goto"

الجدال العام حول "goto" هو أن الشّفرة بدون "goto" هي شفرة أعلى كفاءة.[1] المقالة الشّهيرة التّي أشعلت الجدال الأصلي كانت مقالة ايدجر ديجكسترا (Edsger Dijkstra) (عبارة "goto" اعتبرت مؤذية) ("Go To Statement Considered Harmful") – في آذار عام 1968 ("Communications of the ACM"). لاحظ ديجكسترا أن كفاءة الشفرة كانت متناسبة عكسًا مع عدد "goto" التي يستخدمها المبرمج. وفي عمل آخر، جادل ديجكسترا بأن الشفرة التي لا تحوي "goto" يمكن اثبات صحّتها بشكل أسهل.

الشفرة التي تحوي "goto" من الصعب صياغتها. حيث يجب أن تُستخدم المسافة البادئة لتظهر البنية المنطقيّة، و"goto" لديها تأثير على البنية المنطقيّة. لكن استخدام المسافة البادئة لإظهار البنية المنطقيّة لـ "goto" وهدفها قد يكون صعبًا أو مستحيلًا.

يتغلّب استخدام "goto" على تحسينات المترجم. تعتمد بعض التّحسينات على تدفُّق التحكّم الموجود في عدّة عبارات في البرنامج. يجعل استخدام "goto" بشكل غير متوقّع تحليل التدفّق أكثر صعوبة ويقلّل قدرة المترجم على تحسين الشّفرة. لذلك حتّى وإن كان استخدام "goto" ينتج فعاليّة في مرحلة لغة المصدر، فإنها أيضًا تقلّل الفعاليّة الكليّة عبر إعاقة تحسينات المترجم.

[1] cc2e.com/1785

يجادل مقترحو "goto" غالبًا بأنّها تجعل الشّفرة أسرع وأصغر. لكن الشّفرة التي تحوي "goto" نادرًا ما تكون الأسرع أو الأصغر. مقالة دونالد ناث (Donald Knuth) الرّائعة (البرمجة الهيكليّة مع عبارة "goto") – "structured programming with go to statements" تعطي عدّة أمثلة لحالات يكون فيها استخدام "goto" ينتج شفرة أبطأ وأكبر (knuth 1974).

خلال العمل، فإن "goto" تقود إلى اختراق المبدأ الذي ينص على أن الشفرة يجب أن تتدفّق حصرًا من الأعلى للأسفل. حتى وإن لم تكن "goto" مُحيِّرة عند استخدامها بحذر، عندما توضع في شفرة ما، فهي تنتشر فيها كما ينتشر النّمل الأبيض في بيت عفِن. عند السّماح باستخدام "goto"، هناك سيّئات وإيجابيّات، ولكن من الأفضل عدم السّماح بأي منهما.

بشكل عام، أظهرت التّجارب في العقدين الأخيرين الذين لحقا الإعلان عن مقالة ديجكسترا سخافة إنتاج شفرات مليئة بـ "goto". في إحدى المقالات في محاضرة، ختم بن شنايدرمان (Ben Shneiderman) بأن الدلائل تدعم وجهة نظر ديجكسترا بأنّنا أفضل بدون "goto" (1980)، والعديد من اللّغات الجديدة، متضمنّةً الجافا، لا تحوي "goto".

الجدال لصالح "goto"

يلخّص الجدال لصالح التعليمة "goto" بأنّه مع استخدامها بحذر وفي ظروف خاصّة وليس بشكل عشوائي. معظم الجدالات ضد "goto" هي فقط ضد الاستخدام العشوائي لها. اندلعت المناظرة حول "goto" عندما كانت فورتران "Fortran" هي اللّغة الأكثر رواجًا. وهذه اللغة ليس لديها أي بنى حلقات قابلة للتّقديم، وفي غياب النّصيحة حول برمجة الحلقات باستخدام "goto"، كتب المبرمجون الكثير من الشفرات غير الصّحيحة. هذه الشّفرات بدون شكّ كانت مرتبطة مع إنتاج برامج قليلة الجودة، ولكن لم يكن لها علاقة بالاستخدام الحذر ل "goto" وذلك للتعويض عن الفراغ الكبير بين إمكانيات لغات البرمجة الحديثة.

"goto" موضوعة بشكل جيّد يمكن أن تنفي الحاجة إلى شفرة مكرّرة. تقود الشّفرة المكرّرة إلى مشاكل عدّة إذا كان هناك مجموعتان من الشّفرة معدّلتان بشكل مختلف، وتزيد حجم ملفّات المصدر والملفات القابلة للتّنفيذ. إن التّأثيرات السّيئة لـ "goto" قليلة جدًّا في حالة مخاطر الشفرة المكرّرة.

"goto" مفيدة في الإجرائيّة التي تخصّص الموارد،1 وتقوم بعمليّات على هذه الموارد، ثم تلغي التّخصيص على الموارد. مع "goto"، يمكنك التنظيف باستخدام قسم واحد من الشّفرة. "goto" تقلّل من إمكانيّة نسيانك لإلغاء تخصيص الموارد في كل مكان تكتشف فيه خطأ.

في بعض الحالات، يمكن أن تؤدّي "goto" إلى شفرة أسرع وأصغر. وثّقت مقالة نات (Knuth's 1974) عدّة حالات أنتجت فيها "goto" ربحا منطقيًا.

لا تقصد البرمجة الجيّدة حذف "goto". يقود التفكيك المنهجي، والتّصفية، واختيار بنى التّحكّم تلقائيًا إلى برامج خالية من "goto" في معظم الحالات. الحصول على شفرة بدون "goto" ليس الهدف، بل النّتيجة، والتّركيز على تجنّب "goto" ليس مفيدًا.

فشلت البحوث التي دامت لعقود عن "goto" في توضيح ضررها2. في مقالة لمحاضرة، "ب. أ. شيل" (B. A. Sheil) ختم بأنّ الشّروط غير الواقعيّة، والضّعف في تحليل البيانات، والنّتائج غير الحاسمة فشلت في دعم ما تزعّمه "شنايدرمان" (Shneiderman) وآخرون بأنّ عدد الثّغرات في الشّفرة كان متناسبًا مع عدد "gotos" (1981). لم يختم "شيل" تمامًا بأنّ "goto" فكرة جيّدة- بل ختم بأن الدّلائل التّجريبيّة ضدّها لم تكن حاسمة.

أخيراً، "goto" مشمولة في معظم اللّغات الحديثة، متضمّنةً فيجوال بيسيك، سي++، ولغة آدا (Ada)، أكثر لغة برمجة مُهَندسَة بدقّة في التّاريخ. لغة آدا تم تطويرها بعد أن تطوّر الجدال من الطّرفين حول "goto" إلى أقصى حدوده، وبعد الأخذ بعين الاعتبار كل أطراف هذه القضيّة، مهندسو لغة آدا قرّروا تضمين "goto" فيها.

مناظرة "goto" الزّائفة

إنّ الصّفة الرئيسيّة لمعظم النّقاشات حول "goto" هي مقاربة سطحية للقضيّة. المجادل حول أن " "goto" سيّئة " يقدّم جزء من شفرة صغيرة تستخدم "goto" ثمّ يظهر مدى سهولة أن تعيد كتابة الجزء بدون "goto". وهذا بشكل قطعي يثبت أنه من السهل كتابة شفرة صغيرة بدون "goto".

المجادل عن جانب " لا يمكن العيش بدون "goto" " يقدّم عادةً الحالة التي فيها إلغاء "goto" يقوم بزيادة المقارنات أو التّعقيد لسطر الشفرة. هذا يثبت بشكل رئيسي أن استخدام "goto" في بعض الحالات يسبب مقارنة واحدة أقلّ – وهذا ليس مكسبًا كبيرًا في حواسيب يومنا هذا.

[1] **إشارة مرجعية:** للتفاصيل حول استخدام "goto" في شفرة تخصّص الموارد، انظر الى (معالجة الأخطاء و"goto") في هذا القسم. وانظر أيضًا الى (النّقاش حول معالجة الاستثناءات) في القسم 8-4 "الاستثناءات".

[2] الدّلائل تقترح أنّه فقط بنى التحكّم العشوائيّة هي فقط ما يخفّض اداء المبرمج. هذه الاختبارات لا تعطي أي دليل افتراضي للتأثيرات المفيدة لأي طريقة محدّدة من تدفق بنى التحكّم. _ ب. أ. شيل (B. A. Sheil)

معظم الكتب لا تساعد. حيث تعطي مثالًا صغيرًا لإعادة كتابة شفرة ما بدون "goto" كما لو أن ذلك يغطي الموضوع بأكمله. وهنا مثال صغير لشفرة من مثل تلك الكتب:

مثال لشفرة سي ++ من المفترض أن تكون سهلة لإعادة الكتابة بدون "goto":

```
do {
    GetData( inputFile, data );
    if ( eof( inputFile ) ) {
        goto LOOP_EXIT;
    }
    DoSomething( data );
} while ( data != -1 );
LOOP_EXIT:
```

والكتاب يقوم باستبدال هذه الشفرة بشفرة لا تحوي "goto" بسرعة:

مثال لشفرة سي++ مماثلة للشفرة الأولى، لكن تمت إعادة كتابتها بدون"goto":

```
GetData( inputFile, data );
while ( ( !eof( inputFile ) ) && ( ( data != -1 ) ) ) {
    DoSomething( data );
    GetData( inputFile, data )
}
```

هذا الذّي يسمّى مثالًا "بسيطًا" يحتوي خطأً في الحالة التي فيها البيانات تساوي "1-" قبل دخول الحلقة، الشفرة المترجمة يكشف ال "1-" ويخرج من الحلقة قبل تنفيذ "DoSomething()". تنفّذ الشفرة الأصلية "DoSomething()" قبل اكتشاف ال "1-". كتاب البرمجة الذي حاول أن يظهر مدى سهولة كتابة الشّفرة بدون "goto" ترجم مثاله بطريقة خاطئة. ولكن كاتب هذا الكتاب لا يجب أن يشعر بالسّوء؛ فالكتب الأخرى لديها أخطاء مشابهة. حتّى المحترفون لديهم صعوبات في ترجمة الشّفرة التي تستخدم "goto".

وهنا ترجمة دقيقة للشّفرة بدون "goto":

مثال صحيح لشفرة سي ++ لنفس الشّفرة، تمت إعادة كتابتها بدون "goto":

```
do {
    GetData( inputFile, data );
    if ( !eof( inputFile )) {
        DoSomething( data );
    }
} while ( ( data != -1 ) && ( !eof( inputFile ) ) );
```

حتى مع التّرجمة الصّحيحة للشّفرة، المثال ما يزال غير دقيقًا لأنه يعرض استخدامًا بسيطًا ل "goto". مثل هذا الحالات ليست هي الحالات التي يقوم فيها المبرمجون المفكّرون باختيار "goto" كشكلهم المفضّل في التّحكّم.

سيكون صعبًا في هذا التّاريخ المتأخّر أن نضيف أيّ شيء للجدل النظري حول "goto". ما لا يتطرّق إليه عادةً هو الوضع الذي يكون فيه المبرمج مدركًا كليًّا لبدائل "goto" ويختار "goto" لزيادة القدرة على القراءة والصيانة.

تقدّم الأقسام الآتية حالات يجادل فيها بعض المبرمجون الخبراء من أجل استخدام "goto". أعطت النّقاشات أمثلة عن الشفرة مع "goto" والشفرة المعاد كتابتها بدون "goto" وتقييم المبادلة بين النّسخ.

معالجة الأخطاء و"goto"

يستلزم كتابة شفرة متداخلة بشكل كبير إعطاء الكثير من الانتباه إلى معالجة الأخطاء وتنظيف الموارد عندما يحصل خطأ ما. مثال الشفرة التّالي ينقّي مجموعة من الملفّات. تقوم الإجرائيّة بتنقية مجموعة من الملفّات أولًا، ثم تجد كل ملف، تفتحه، وتكتب فوقه ثمّ تمحوه. والإجرائيّة تتحقّق من الأخطاء في كل خطوة:

شفرة فيجوال بيسيك مع "goto" تعالج الأخطاء وتنظّف الموارد:

```
' This routine purges a group of files.
Sub PurgeFiles( ByRef errorState As Error_Code )
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer

    MakePurgeFileList( fileList, numFilesToPurge )
    errorState = FileStatus_Success
    fileIndex = 0
    While ( fileIndex < numFilesToPurge )
        fileIndex = fileIndex + 1
        If Not ( FindFile( fileList( fileIndex ), fileToPurge ) ) 
Then
            errorState = FileStatus_FileFindError
            GoTo END_PROC
        End If                              ← استخدمت goto

        If Not OpenFile( fileToPurge ) Then
            errorState = FileStatus_FileOpenError
            GoTo END_PROC
        End If                              ← استخدمت goto

        If Not OverwriteFile( fileToPurge ) Then
            errorState = FileStatus_FileOverwriteError
            GoTo END_PROC
        End If                              ← استخدمت goto

        if Not Erase( fileToPurge ) Then
            errorState = FileStatus_FileEraseError
            GoTo END_PROC
        End If                              ← استخدمت goto
    End While

END_PROC:                                   ← هذه هي علامة goto
    DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

هذه التعليمة نموذجية في الظروف التي يقرر فيها المبرمجون الخبراء أن يستخدموا "goto" تظهر حالات مشابهة عندما تحتاج إجرائية أن تخصص وتنظف الموارد مثل روابط قاعدة البيانات، الذاكرة والملفات المؤقتة.

في تلك الحالات ينسخ بديل "gotos" عادة الشفرة لينظف الموارد. في مثل هذه الحالات، ربما يقارن المبرمج سوء "goto" مقابل صداع الحفاظ على شفرة منسوخة ويقرر أن "goto" أقل سوءًا.

تستطيع أن تعيد كتابة الإجرائية السابقة بطريقتين لتجنّب "goto"، وكليهما يتطلّب مقايضات. خطط إعادة الكتابة الممكنة تكون:

إعادة الكتابة مع تعابير "if" المتداخلة. لتعيد الكتابة بتعابير "if" المتداخلة، أدخل تعابير "if" بحيث يُنفَّذ كل منها فقط إذا نجح الاختبار السابق. هذه هي منهجية البرمجة القياسية للكتب لإزالة "gotos". هنا إعادة كتابة للإجرائية التي تستخدم المنهجية القياسية[1]:

شفرة فيجوال بيسيك تتجنب استخدام goto باستخدام if المتداخلة

```
' This routine purges a group of files.
Sub PurgeFiles( ByRef errorState As Error_Code )
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer

    MakePurgeFileList( fileList, numFilesToPurge )

    errorState = FileStatus_Success
    fileIndex = 0
    While ( fileIndex < numFilesToPurge And errorState = FileStatus_Suc-
cess )

        fileIndex = fileIndex + 1
        If FindFile( fileList( fileIndex ), fileToPurge ) Then
            If OpenFile( fileToPurge ) Then
                If OverwriteFile( fileToPurge ) Then
                    If Not Erase( fileToPurge ) Then
                        errorState = FileStatus_FileEraseError
                    End If
                Else ' couldn't overwrite file
                    errorState = FileStatus_FileOverwriteError
                End If
            Else ' couldn't open file
                errorState = FileStatus_FileOpenError
            End If
        Else ' couldn't find file
            errorState = FileStatus_FileFindError
        End If
    End While
    DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

اختبار while تمّ تغييره لإضافة اختبار errorState

هذا السّطر بعيد 13 سطراً عن عبارة if التي يمثّلها.

بالنسبة للناس التي اعتادت البرمجة بدون "gotos"، من الممكن أن تكون هذه الشفرة أسهل للقراءة من نسخة "goto"، وإذا استخدمتها، لن يكون عليك أن تواجه استجوابًا من جماعة "goto" الخرقاء.

[1] **إشارة مرجعية:** هذه الإجرائية يمكن أيضًا إعادة كتابتها باستخدام " break" وبدون "goto". للتفاصيل حول هذه المنهجية، انظر "الخروج المبكّر من الحلقات" في القسم 16-2.

السّلبية الرئيسية لنهج "if" المتداخلة هي أن التداخل عميق جدًا.[1] لكي تفهم الشفرة، عليك أن تحتفظ بالمجموعة الكاملة لتعابير "if" المتداخلة في ذهنك دفعة واحدة. وأكثر من ذلك، المسافة بين شفرة معالجة الخطأ والشفرة التي تستدعيها تكون كبيرة جدًا، مثلًا: الشفرة التي تعين "errorState" لـ "FileStatus_FileFindError" تبعد "13" سطراً عن عبارة "if" التي تستدعيها.

مع نسخة "goto" لا يبعد أي تعبير أكثر من أربعة أسطر عن الحالة التي تستدعيه، ولا تحتاج أن تحتفظ بالبنية الكاملة في ذهنك دفعة واحدة. تستطيع أساسًا أن تتجاهل أي حالات سابقة ناجحة وتركز على العملية التالية. في هذه الحالة تكون نسخة "goto" أكثر قابلية للقراءة والدعم من نسخة "if" المتداخلة.

إعادة الكتابة مع متغيّر حالة *(a status variable)*. لتعيد الكتابة بمتغيّر حالة، أوجد متغير يبيّن إذا كانت الإجرائية بحالة خطأ. في هذه الحالة، تستخدم الإجرائية مسبقًا متغير الحالة "errorState"، وهكذا تستطيع أن تستخدم ذلك.

شفرة فيجول بيسك تتجنّب استخدام "goto" باستخدام متغيّر حالة

```
 ' This routine purges a group of files.
 Sub PurgeFiles( ByRef errorState As Error_Code )
     Dim fileIndex As Integer
     Dim fileToPurge As Data_File
     Dim fileList As File_List
     Dim numFilesToPurge As Integer

     MakePurgeFileList( fileList, numFilesToPurge )

     errorState = FileStatus_Success
     fileIndex = 0
     While ( fileIndex < numFilesToPurge ) And ( errorState = FileStatus_Suc-
cess )
         fileIndex = fileIndex + 1

         If Not FindFile( fileList( fileIndex ), fileToPurge ) Then
             errorState = FileStatus_FileFindError
         End If
         If ( errorState = FileStatus_Success ) Then
             If Not OpenFile( fileToPurge ) Then
                 errorState = FileStatus_FileOpenError
             End If
         End If
         If ( errorState = FileStatus_Success ) Then
             If Not OverwriteFile( fileToPurge ) Then
                 errorState = FileStatus_FileOverwriteError
             End If
         End If
         If ( errorState = FileStatus_Success ) Then
             If Not Erase( fileToPurge ) Then
                 errorState = FileStatus_FileEraseError
             End If
         End If
     End While
     DeletePurgeFileList( fileList, numFilesToPurge )
 End Sub
```

اختبار while تمّ تغييره لإضافة اختبار errorState

متغير الحالة يتمّ اختباره

متغير الحالة يتمّ اختباره

متغير الحالة يتمّ اختباره

[1] **إشارة مرجعية:** للمزيد من التفاصيل عن مسافة البدء وغيرها من قضايا تنسيق الشفرة، انظر الفصل 31 "المظهر والأسلوب". لتفاصيل عن مراحل التداخل انظر الفصل 19-4، "ترويض التداخل العميق الخطير".

إيجابية مقاربة متغير الحالة هي أنها تتحاشى بنى "if-then-else" المتداخلة بعمق لإعادة الكتابة الأولى وبالتالي تكون أسهل للفهم. وأيضاً يضع الإجراء التالي لاختبار "if-then-else" بمكان أقرب إلى الاختبار أكثر مما تفعل منهجية "if" المتداخلة ويتحاشى كليًا عبارات "else".

يتطلب فهم نسخة "if" المتداخلة بعض ألعاب القوى الذهنية. نسخة متغير الحالة أسهل للفهم لأنها تحاكي عن قرب الطريقة التي يفكر بها الناس بالمشكلة. جد الملف. إذا كان كل شيء جيدًا، افتح الملف. إذا بقي كل شيء جيدًا، زد في كتابة الملف، وهكذا...

الشيء السلبي في هذه المنهجية هو أن استخدام متغيرات الحالة ليس شائعًا كتمرين كما يجب أن يكون. وثق استخدامهم بشكل كامل أو أن بعض المبرمجين قد لا يفهموا عملك. في هذا المثال، استخدام الأنواع التعدادية المسماة جيدًا يساعد بشكل كبير.

إعادة الكتابة باستخدام "try-finally". تزود بعض اللغات بما فيها فيجوال بيسيك وجافا تعبير "try-finally" الذي من الممكن أن يُستخدم لتنظيف الموارد في ظل شروط الخطأ. لإعادة الكتابة باستخدام منهج "try-finally"، ضمِّن الشفرة التي لولا ذلك كانت ستحتاج أن تدقق الأخطاء داخل قسم "try"، وضع شفرة التنظيف داخل قسم "finally". تحدّد شفرة "Try" مجال معالجة الاستثناء، وتنجز شفرة "finally" تنظيف أي موارد. ستُستدعى شفرة "Finally" دائمًا بغض النظر فيما إذا كان الاستثناء مطروحًا وبغض النظر إذا كانت إجرائية "PurgeFiles()" تلتقط أي استثناء مرمي.

شفرة فيجوال بيسيك تتجنّب "goto" باستخدام "try-finally":

```
' This routine purges a group of files.
    ' Exceptions are passed to the caller.
Sub PurgeFiles()
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer
    MakePurgeFileList( fileList, numFilesToPurge )
    Try
        fileIndex = 0
        While ( fileIndex < numFilesToPurge )
            fileIndex = fileIndex + 1
            FindFile( fileList( fileIndex ), fileToPurge )
            OpenFile( fileToPurge )
            OverwriteFile( fileToPurge )
            Erase( fileToPurge )
        End While
    Finally
        DeletePurgeFileList( fileList, numFilesToPurge )
    End Try
End Sub
```

تفترض هذه المنهجية أن كل استدعاءات التوابع ترمي الاستثناءات أكثر مما تعيد شفرات الخطأ.

إيجابية منهجية "try-finally" هي أنها أبسط من منهجية "goto" ولا تستخدم "gotos" وأيضاً تتحاشى بنى "if-then-else" المتداخلة بعمق.

قيود منهجية "try-finally" هي أنها يجب أن تُنفَّذ باستمرار من خلال قاعدة شفرة. إذا كانت الشفرة السابقة جزءًا من قاعدة شفرة استخدمت شفرات خطأ بالإضافة إلى الاستثناءات، ستكون شفرة الاستثناء مطلوبة لوضع شفرات الخطأ لكل خطأ محتمل، وهذه الحاجة ستجعل الشفرة بنفس تعقيد المنهجيات الأخرى.

مقارنة المنهجيات

كل من الطرق الأربعة لديها شيء ليقال عنها. تتحاشى منهجية "goto" التداخل العميق والاختبارات غير الضرورية ولكن طبعًا لديها "gotos".[1] تتحاشى منهجية "if" المتداخلة "gotos" ولكنها متداخلة بعمق وتعطي صورة مبالغ بها عن التعقيد المنطقي للإجرائية. تتحاشى منهجية متغيّر الحالة "goto" والتداخل العميق لكن تقدّم اختبارات إضافية. تتحاشى منهجية "try-finally" كل من "gotos" والتداخل العميق ولكنها غير متوفرة في جميع اللغات.

منهجية "try-finally" هي الأكثر سهولة في اللغات التي تزود "try-finally" وفي قواعد الشفرة التي لم توحد المقاييس مسبقًا على منهجية أخرى. إذا لم تكن "try-finally" خيارًا، منهجية متغيّر الحالة مفضلة قليلًا على منهجيات "if" المتداخلة و"goto" لأنها أكثر قابلية للقراءة وتصوغ المسألة بشكل أفضل، لكن ذلك لا يجعلها المنهجية الأفضل في كل الظروف.

يعمل أي من هذه التقنيات جيدًا عند تطبيقه باستمرار على كل الشفرة في مشروع. خذ بالاعتبار كل المنهجيات، ثم اتخذ قرار أي طريقة تفضل للمشروع.

"gotos" ومشاركة الشفرة في عبارة "else"

أحد مواقف التحدي التي سيستخدم فيها بعض المبرمجون "goto" هي الحالة التي يكون لديك فيها اختبارين شرطيين وعبارة "else" وتريد أن تنفذ شفرة في أحد الشروط وفي عبارة "else". هنا مثال عن حالة من الممكن أن تقود شخص ل "goto":



[1] إشارة مرجعية: للحصول على القائمة الكاملة من التّقنيّات التي يمكن تطبيقها على أوضاع مثل هذه، انظر "ملخّص لتقنيّات تقليل التّداخل العميق" في القسم 19-4.

مثال سي++ لشفرة مشتركة في عبارة "else" مع "goto":

```
if ( statusOk ) {
    if ( dataAvailable ) {
        importantVariable = x;
        goto MID_LOOP;
    }
}
else {
    importantVariable = GetValue();

    MID_LOOP:

    // lots of code
    ...
}
```

هذا مثال جيد لأنه غير مستقيم منطقياً – من المستحيل تقريباً أن تقرأه كما يمثل. ومن الصعب أن تعيد كتابته بشكل صحيح بدون "goto". إذا كنت تعتقد أنّك تعيد كتابته بشكل صحيح بدون "goto"، اسأل شخصاً ما ليراجع شفرتك. عدة مبرمجين خبراء أعادوا كتابته بشكل غير صحيح.

تستطيع أن تكتب الشّفرة بعدّة طرق. تستطيع أن تنسخ الشّفرة، وتضع الشفرة العامّة في إجرائية ما وتستدعيها من مكانين، أو أن تعيد اختبار الشّروط. في معظم اللّغات، إعادة الكتابة تكون أبسط وأكبر بشكل صغير جدّاً من الأصل. ولكنّها تكون قريبة منها بشكل كبير. ما لم تكن الشفرة في عقدة صعبة، أعد كتابتها بدون التفكير في فعاليّتها.

أفضل إعادة كتابة تكون بوضع الجزء "//lots of code" في إجرائيّتها الخاصة. ثمّ تستطيع أن تستدعي الإجرائيّة من الأمكنة التي لولا ذلك كنت ستستخدمها كأصول أو وجهات لـ "goto" وتحافظ على البنية الأصلية للشرطيّة. وهنا كيف يبدو ذلك:

مثال سي++ لشفرة مشتركة في عبارة "else" عبر وضع شفرة شائعة في إجرائيّة

```
if ( statusOk ) {
    if ( dataAvailable ) {
        importantVariable = x;
        DoLotsOfCode( importantVariable );
    }
}
else {
    importantVariable = GetValue();
    DoLotsOfCode( importantVariable );
}
```

عادةً، كتابة إجرائيّة جديدة هي المنهجية الأفضل. وأحياناً، ليس عمليّاً أن تضع شفرة منسوخة في إجرائيّتها الخاصّة. في هذه الحالة، تستطيع أن تلتفّ حول الحل غير العمليّ وذلك بإعادة التّركيب للشرطيّة وبذلك تبقى الشّفرة في نفس الإجرائيّة من دون أن تضعها في إجرائيّة جديدة:



مثال سي++ لشفرة مشتركة في عبارة "else" بدون "goto":

```
if ( ( statusOk && dataAvailable ) || !statusOk ) {
    if ( statusOk && dataAvailable ) {
        importantVariable = x;
    }
    else {
        importantVariable = GetValue();
    }

    // lots of code
    ...
}
```

هذه ترجمة ميكانيكيّة صحيحة للمنطق في نسخة "goto".[1] وتختبر "statusOK" مرّتين إضافيّتين و"dataAvailable" مرّة إضافيّة، ولكن تكون الشفرة متعادلة. إذا كانت إعادة اختبار الشرطيّات تزعجك، لاحظ أن قيمة "statusOK" لا تحتاج إلى الاختبار مرّتين في اختبار "if" الأوّل. وكذلك يمكنك تجاهل اختبار "dataAvailable" في اختبار "if" الثّاني.

ملخّص إرشادات لاستخدام "goto"

استخدام "goto" هو كأمر متعلّق بالدّين. مبدأي هو أنّه في اللّغات الحديثة، تستطيع أن تستبدل بسهولة تسع من أصل عشر "goto" بتراكيب تسلسليّة معادلة لها. في هذه الحالات البسيطة، يجب أن تستبدل "goto" كالعادة. في الحالات الصّعبة، تستطيع أن تطرد "goto" في تسع من أصل عشر حالات أيضاً: يمكنك تقسيم الشّفرة إلى إجرائيّات أصغر، استخدم "try-finally"، استخدم "if" المتداخلة، اختبر وأعد اختبار متحوّل حالة، أو إعادة صياغة الشّرطيّة. إزالة "goto" أصعب في هذه الحالة، لكنّه تمرين ذهني جيّد والتّقنيّات المستخدمة في هذا القسم تعطيك الأدوات للقيام بذلك.

في الحالة الوحيدة المتبقيّة من أصل 100 حالة يكون فيها "goto" حلّاً منطقيًّا للمشكلة، وثّقها بوضوح واستخدمها. إذا كنت ترتدي حذاءً مطريًّا، لا داعي أن تلتفّ حول المربّع السكني فقط لتتفادى بركة وحل. لكن انتبه الى المنهجيات بدون "goto" المقترحة من مبرمجين آخرين. لربما يرون شيئاً لا تراه أنت.

هذا ملخص الإرشادات لاستخدام "goto":

- استخدم "goto" لتُشابه بنى التحكّم المركّبة في اللغات التي لا تدعمها بشكل مباشر. عندما تفعل ذلك، ناقشها بالضّبط. لا تسئ استخدام المرونة الإضافيّة التي تعطيها "goto" لك.
- لا تستخدم "goto" عندما تتوافر لديك بنى مضمّنة معادلة لها.

[1] **إشارة مرجعية** منهجية أخرى لهذه المشكلة هي استخدام جدول القرارات. للتفاصيل، انظر الفصل 18 "الطرق جدولية القيادة"

- قم بقياس أداء أي "goto" مســتخدمة لزيادة الفاعليّة. [1]في معظم الحالات، تســتطيع إعادة كتابة الشـفرة بدون "goto" لتحسـين القراءة وبدون أيّ فقدان في الفعاليّة. إذا كانت حالتك هي الاسـتثناء، وثّق التّحسين في الفعاليّة الذي قامت به لكيلا يقوم كارهي "goto" بحذفها عندما يرونها.
- حدّد نفسك باستخدام علامة "goto" واحدة في كل إجرائيّة إلّا إذا كنت تستخدمها مكان بنى التّحكّم.
- حدّد نفسك بـ "goto" الذّاهبة إلى الأمام وليس إلى الخلف إلّا إذا كنت تستخدمها مكان بنى التّحكّم.
- تأكّد من أن كل علامات "goto" قد تمّ اسـتخدامها. العلامات غير المسـتخدمة قد تكوّن كشـفًا لشـفرة مفقودة، لأن الشّفرة تذهب إلى العلامات. إذا كانت العلامات غير مستخدمة، احذفها.
- تأكد من أن "goto" لا تنشئ شفرة لا يمكن الوصول إليها.
- إذا كنت مديرًا، تبنّى المبدأ القائل بأن معركة حول "goto" واحدة ليست بقيمة خسارة الحرب. إذا كان المبرمج عالمًا بالبدائل وعلى استعداد للجدال، استخدام "goto" ربما يكون جيّداً.

4.17 وجهة نظر في تراكيب التحكم الغير عادية

في وقت ما أو آخر، ظن أحدهم أن كل من بنى التحكم التالية فكرة جيدة:

- الاستخدام الغير مقيد لـ "gotos".
- القدرة على حوسبة هدف "goto" ديناميكيًا والقفز إلى الموقع المحوسب.
- القدرة على استخدام "goto" للقفز من وسط إجرائية إلى وسط إجرائية أخرى.
- القدرة على استدعاء إجرائية برقم سطر أو بعلامة سمحت للتنفيذ بالبدء في مكان ما وسط الإجرائية.
- القدرة على الحصول على شفرة تشغيل البرنامج بسرعة ثم تنفيذ الرمز الذي كتبه للتو.

في زمن ما، اعتُبرت كل واحدة من هذه الأفكار مقبولة أو حتى مرغوبة، بالرغم من أنها تبدو جميعها الآن قديمة وخطيرة. لقد تقدّم مجال تطوير البرامج كثيرًا من خلال حصر ما يستطيع أن يفعله المبرمجون بشفراتهم. وبالنتيجة، أرى بنى تحكم غير تقليدية بارتياب كبير. أشك أن غالبية التراكيب في هذا الفصل ستجد طريقها بالنهاية إلى كومة النفايات للمبرمج بالترافق مع علامات "goto" المحوسبة، نقاط الدخول المتغيرة للإجرائية، شفرة معدلة ذاتيًا وتراكيب أخرى فضلت المرونة والسهولة على التركيب والقدرة على إدارة التعقيد.

مصادر إضافية

[1] **إشارة مرجعية:** لتفاصيل حول تحسين الفعاليّة، انظر الفصل 25 "استراتيجيات ضبط الشفرة" والفصل 26 "تقنيّات ضبط الشفرة"

الموارد الآتية تتطرق أيضاً لبنى التحكم الغير عادية:[1]

أوامر الإعادة "Returns"

Fowler، Martin. Refactoring improving the Design of Existing Code. Reading، MA: Addison-Wesley، 1991 .

في وصف عملية إعادة البناء التي تسمى "استبدال الشرط المتداخل مع بنود الحماية"، يقترح فاولر استخدام الإعادة المتعددة من إجرائية للتقليل من التداخل في مجموعة من عبارات "if". يجادل فاولر بأن الإعادة المتعددة هي وسيلة مناسبة لتحقيق قدر أكبر من الوضوح، وأنه لا يوجد ضرر ناجم عن وجود إعادات متعددة من إجرائية.

"Gotos"

تحتوي هذه المقالات على كل المناظرات حول "goto"[2]. تظهر بقوة من حين لآخر في معظم أماكن العمل، الكتب، والمجلات، لكنك لن تسمع بأي شيء لم يُكتشف كليًا من عشرين سنة مضت.

Dijkstra، Edsger. “Go To Statement Considered Harmful.” Communications of the ACM 11، no. 3 (March 1968): 147–48،

متوفر أيضا من:

www.cs.utexas.edu/users/EWD/.

هذه هي الرسالة الشهيرة التي وضع فيها ديجكسترا عود الثقاب على الورق وأشعل أحد أطول الخلافات في تطوير البرمجيات.

Wulf، W. A. “A Case Against the GOTO.” Proceedings of the 25th National ACM Conference، August 1972: 791–97.

كانت هذه الورقة حجة أخرى ضد الاستخدام العشوائي لـ "gotos". قال "Wulf" أنه إذا كانت لغات البرمجة توفر بنى تحكم كافية، فإن "gotos" سيصبح غير ضروري إلى حد كبير. منذ عام 1972، عندما تم كتابة الورقة، أثبتت لغات برمجة مثل سي++، جافا، و"فيجوال بيسيك" صحة ما قال "Wulf".

Knuth، Donald. “Structured Programming with go to Statements” 1974. In Classics in Software Engineering، edited by Edward Yourdon. Englewood Cliffs، NJ: Yourdon Press، 1979 .

[1] cc2e.com/1792

[2] cc2e.com/1799

هذه الورقة الطويلة لا تتعلق كليًا بـ "gotos"، ولكنها تتضمن حشدًا من الأمثلة البرمجية التي يتم جعلها أكثر فاعلية من خلال القضاء على "gotos" ومجموعة أخرى من الأمثلة البرمجية التي يتم إجراؤها بكفاءة أكثر بإضافة "gotos".

Rubin، Frank. "'GOTO Considered Harmful' Considered Harmful." Communications of the ACM 30، no. 3 (March 1987): 195–96.

في هذا الخطاب إلى رئيس التحرير، يؤكد روبن أن البرمجة بدون "goto" قد كلفت الشركات "مئات الملايين من الدولارات". ثم يقدم جزءًا قصيرًا من الشفرة تستخدم "goto" ويجادل بأنه متفوق على البدائل التي لا تستخدم "goto".

الرد الذي أحدثته رسالة "Rubin" كان أكثر إمتاعًا من الرسالة نفسها. لمدة خمسة أشهر، نشرت "Communications of the ACM(CACM)" رسائل قدمت نسخ مختلفة لبرنامج "Rubin" المؤلف من سبعة أسطر. كانت الرسائل مقسومة بالتساوي بين أولئك المدافعين عن "gotos" وأولئك الذين يتحدثون عنها بقسوة. اقترح القراء تقريبا "17" إعادة كتابة مختلفة، والشفرة المعاد كتابتها غطت تمامًا طيف منهجيات تجنّب "gotos". كتب محرر "CACM" أن الرسالة قد أحدثت ردًا أكبر بكثير من أي قضية أخرى أُخذت بالاعتبار على صفحات "CACM".

من أجل الرسائل التّابعة شاهد:

- Communications of the ACM 30، no. 5 (May 1987): 351–55.
- Communications of the ACM 30، no. 6 (June 1987): 475–78.
- Communications of the ACM 30، no. 7 (July 1987): 632–34.
- Communications of the ACM 30، no. 8 (August 1987): 659–62.
- Communications of the ACM 30، no. 12 (December 1987): 997،1085.

Clark، R. Lawrence، "A Linguistic Contribution of GOTO-less Programming," Datamation، December 1973.[1]

تجادل هذه الورقة الكلاسيكية باستهزاء عن استبدال عبارة "GO To" بعبارة "Come From". كما أعيدت طباعتها في طبعة أبريل 1974 من ("Communications of the ACM").

[1] cc2e.com/1706

قائمة التحقق: بنى التحكم غير العادية 1

أمر الإعادة "Return"

- هل تستخدم كل إجرائية "return" فقط عند الضرورة؟
- هل يعزز "return" سهولة القراءة؟

العوديّة "Recursion"

- هل تحتوي إجرائية العوديّة شفرة لوقف العوديّة؟
- هل تستخدم الإجرائية عدّاد سلامة لتضمن توقفها؟
- هل العوديّة محدودة بإجرائية واحدة؟
- هل عمق إجرائية العوديّة ضمن الحدود المفروضة من قبل حجم مكدّس البرنامج؟
- هل العوديّة هي الطريقة الأفضل لتنفيذ الإجرائية؟ هل هي أفضل من التعليمات التكرارية البسيطة؟

"Goto"

- هل تُستخدم "gotos" فقط كملجأ أخير، ثم فقط لتجعل الشفرة أكثر سهولة للقراءة وأكثر قابلية للصيانة؟
- إذا كانت "goto" تُستخدم من أجل الفعالية، هل الكسب في الفعالية يُقاس أو يُوثّق؟
- هل "gotos" محدودة بعلامة واحدة لكل إجرائية؟
- هل جميع "gotos" تذهب إلى الأمام وليس إلى الخلف؟
- هل تُستخدم كل علامات "goto"؟

نقاط مفتاحية

- الإعادات المتعدّدة يمكنها تحسين القدرة على قراءة و صيانة الإجرائيّة، ويمكنها منع المنطق المتداخل بعمق. لكن مع ذلك يجب أخذ الحيطة باستخدامها.
- العوديّة توفّر حلولًا أنيقة لمجموعة صغيرة من المشاكل. استخدمها بحذر أيضًا.
- في بعض الحالات، "gotos" هي الطريقة الأفضل لكتابة شفرة مقروءة وقابلة للصيانة. هذه الحالات نادرة. استخدم "goto" فقط كملجأ أخير.

[1] cc2e.com/1713

الفصل 18

الطرق جدوليّة القيادة

المحتويات[1]

مواضيع ذات صلة

- إخفاء المعلومات: "خبئ الأسرار (إخفاء المعلومات)" في القسم 5. 3
- تصميم الصفوف: الفصل 6
- استخدام جداول القرارات لتحل محل المنطق المعقد: في القسم 19. 1
- بدّل جداول البحث مكان التعابير المعقدة: في القسم 26. 1

الطرق جدولية القيادة هي مخطط يمكّنك من البحث عن المعلومة في جدول بدلاً من استخدام العبارات المنطقية (`if` و`case`) لمعرفتها. فعلياً أي شيء يمكنك أن تختاره باستخدام العبارات المنطقية، يمكنك أن تختاره بالجداول بدلاً. في الحالات البسيطة، العبارات المنطقية أسهل وأكثر مباشرة. لكن كلما أصبحت سلسلة المنطق أعقد، تصبح الجداول جذابة أكثر.

إذا كنت من قبل متآلفاً مع الطرق جدولية القيادة، يمكن أن يكون هذا الفصل مجرد مراجعة. في تلك الحالة، يمكنك أن تتفحص "مثال تنسيق-الرسالة-المرن" في القسم 18. 2 لترى بمثال جيد كيف أنه لا يكون التصميم غرضي التوجه بالضرورة أفضل من أي نوع آخر من التصاميم فقط لكونه غرضي التوجه، بعدها يمكنك أن تنتقل إلى النقاش حول قضايا التحكم العامة في الفصل 19.

[1] cc2e.com/1865

18. 1 اعتبارات عامة في استخدام الطرق جدولية القيادة



باستخدامها في الظروف المناسبة، الشفرة جدولية القيادة تكون أبسط من المنطق المعقد، وأسهل للتعديل، وأكثر كفاءة. افترض أنك تريد أن تصنف المحارف إلى أحرف وعلامات ترقيم وأرقام؛ قد تستخدم سلسلة معقدة من المنطق كهذه:

```
مثال جافل عن استخدام المنطق المعقد لتصنيف محرف
if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||
     ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {
    charType = CharacterType.Letter;
}
else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||
          ( inputChar == '.' ) || ( inputChar == '!' ) ||
          ( inputChar == '(' ) || ( inputChar == ')' ) ||
       ( inputChar == ':' ) || ( inputChar == ';' ) ||
          ( inputChar == '?' ) || ( inputChar == '-' ) ) {
    charType = CharacterType.Punctuation;
}
else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {
    charType = CharacterType.Digit;
}
```

إذا استخدمت جدول بحث بدلاً، ستخزن نمط كل محرف في مصفوفة يتم الوصول إلى بياناتها عبر رقم (شفرة) المحرف. قطعة الشفرة المعقدة التي رأيتها للتو ستبدل بهذا:

```
مثال جافل عن استخدام جدول بحث لتصنيف محرف

charType = charTypeTable[ inputChar ];
```

هذه القطعة تفترض أنك هيئت المصفوفة charTypeTable مسبقاً. أنت تسند معرفة برنامجك إلى بياناته بدلاً من منطقهِ- أي إلى جدول بدلاً من اختبارات if المنطقية.

قضيتين في استخدام الطرق جدولية القيادة



عندما تستخدم الطرق جدولية القيادة، عليك أن تحل قضيتين. أولاً عليك أن تجيب على السؤال "كيف تبحث عن المداخل في الجدول". يمكنك أن تستخدم بعض البيانات للنفاذ إلى الجدول مباشرة. إذا احتجت أن تصنف البيانات شهرياً، على سبيل المثال، الدخول إلى جدول الشهور يكون مباشراً. بإمكانك استخدام مصفوفة بفهارس من 1 إلى 12.

تكون بعض البيانات الأخرى غير ملائمة مطلقاً للاستخدام في الدخول المباشر إلى مداخل الجدول. إذا كنت تريد أن تصنف البيانات عن طريق "رقم الضمان الاجتماعي"، على سبيل المثال، لن تتمكن من استخدام رقم

الضمان الاجتماعي كمفتاح للدخول إلى الجدول مباشرة إلا إذا استطعت أن تدفع كلفة تخزين 999-99-9999 مدخل في جدولك. إنك مضطر لاستخدام نهج أكثر تعقيداً. هنا لائحة من الطرق للوصول إلى مدخل في جدول:

- الدخول المباشر
- الدخول المفهرس
- الدخول بدرجة الدرج

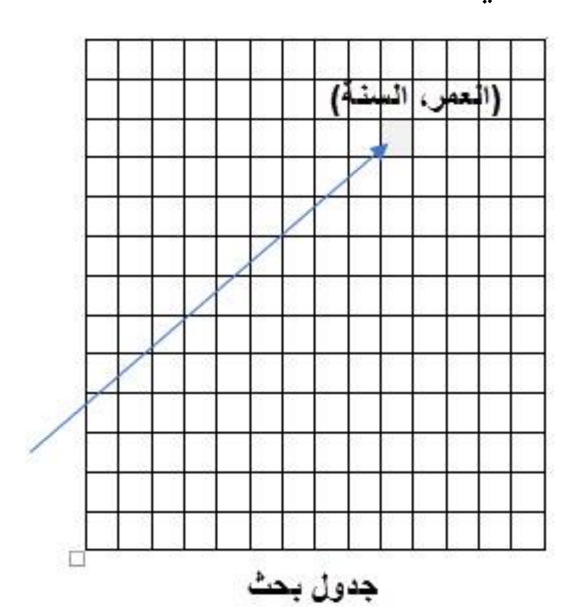
كل من أنواع الدخول هذه مشروحة بتفصيل أكثر في الأقسام الفرعية لاحقاً في هذا الفصل.

نقطة مفتاحية

القضية الثانية التي يجب أن تعالجها إذا أردت استخدام طريقة جدولية القيادة هي ما الذي ينبغي أن يُخزَّن في الجدول. في بعض الحالات، نتيجة البحث في الجدول هي بيانات. إذا كانت الحالة كذلك، تستطيع أن تخزن البيانات في الجدول. في حالات أخرى، نتيجة البحث في الجدول هي عمل. في حالة كهذه، تستطيع أن تخزن الشفرة التي تعبر عن العمل أو، في بعض اللغات، تستطيع أن تخزن مرجع إلى الإجرائية التي تحقق العمل. في كلتا الحالتين، سيصبح الجدول أكثر تعقيداً.

18. 2 جداول الدخول المباشر

تحل جداول الدخول المباشر محل بنيان التحكم المنطقي المعقد، مثل كل جداول البحث. إنها ذات "دخول مباشر" لأنه لا يتوجب عليك القفز فوق العراقيل لتجد المعلومات التي تريدها في الجدول. كما يشير الشكل 18-1، تستطيع أن تلتقط المدخل الذي تريد مباشرة.



الشكل 18-1 كما يشير الاسم، جدول الدخول المباشر يسمح لك أن تصل إلى عنصر الجدول الذي يهمك مباشرة.

مثال الأيام-في-الشهر

لنفرض أنك تريد أن تحدد عدد الأيام لكل شهر (لننسى السنين الكبيسة، لمصلحة النقاش). أن تكتب عبارة if ضخمة، هي بالتأكيد طريقة غير متقنة للقيام بهذا العمل:

مثال فيجوال بيسك عن طريقة غير متقنة لتحديد عدد الأيام في شهر

```
If ( month = 1 ) Then
    days = 31
ElseIf ( month = 2 ) Then
    days = 28
ElseIf ( month = 3 ) Then
    days = 31
ElseIf ( month = 4 ) Then
    days = 30
ElseIf ( month = 5 ) Then
    days = 31
ElseIf ( month = 6 ) Then
    days = 30
ElseIf ( month = 7 ) Then
    days = 31
ElseIf ( month = 8 ) Then
    days = 31
ElseIf ( month = 9 ) Then
    days = 30
ElseIf ( month = 10 ) Then
    days = 31
ElseIf ( month = 11 ) Then
    days = 30
ElseIf ( month = 12 ) Then
    days = 31
End If
```

أن تضع البيانات في جدول هو طريقة أسهل وأكثر قابلية للتعديل لإنجاز نفس الوظيفة. في مايكروسوفت فيجوال بيسيك، عليك أولاً أن تهيئ الجدول:

مثال فيجوال بيسك عن طريقة بارعة لتحديد عدد الأيام في شهر

```
' Initialize Table of "Days Per Month" Data
Dim daysPerMonth() As Integer = _
{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

الآن بدلاً من استخدام عبارة if طويلة، تستطيع فقط أن تستخدم دخول بسيط إلى مصفوفة لتجد عدد الأيام في شهر:

مثال فيجوال بيسك عن طريقة بارعة لتحديد عدد الأيام في شهر (تابع)

```
days = daysPerMonth( month-1 )
```

إذا أردت أن تحسب حساب السنة الكبيسة في إصدار لجدول البحث، ستبقى الشفرة بسيطة، بافتراض أن () LeapYearIndexلها قيمة إما 0 أو 1

مثال فيجوال بيسك عن طريقة بارعة لتحديد عدد الأيام في شهر (تابع)

```
days = daysPerMonth( month-1, LeapYearIndex() )
```

في إصدار عبارة if، سلسلة ifs الطويلة ستنمو على نحو متساوٍ بطريقة أكثر تعقيداً، إذا تم تضمين السنة الكبيسة.

تحديد عدد الأيام في الشهر هو مثال ملائم لأنك تستطيع استخدام المتحول month لتبحث عن مدخل في الجدول. تستطيع غالباً استخدام القيمة التي تحكمت بالكثير من عبارات if، لتصل إلى الجدول مباشرة.

مثال معدلات الضمان

لنفترض أنك تكتب برنامج لحساب معدلات الضمان الصحي، ولديك معدلات تتنوع تبعاً للعمر والجنس والحالة الزوجية وإن كان الشخص مدخناً. إذا أردت أن تكتب بنيان التحكم المنطقي للمعدلات، ستحصل على شيء ما يشبه هذا:



مثال جافا عن طريقة غير متقنة لتحديد معدل الضمان

```
if ( gender == Gender.Female ) {
   if ( maritalStatus == MaritalStatus.Single ) {
      if
      ( smokingStatus == SmokingStatus.NonSmoking ) {
         if ( age < 18 ) {
            rate = 200.00;
         }
      else if ( age == 18 ) {
            rate = 250.00;
         }
         else if ( age == 19 ) {
            rate = 300.00;
         }
         ...
         else if ( 65 < age ) {
            rate = 450.00;
         }
      }
      else {
         if ( age < 18 ) {
            rate = 250.00;
      }
      else if ( age == 18 ) {
         rate = 300.00;
      }
      else if ( age == 19 ) {
         rate = 350.00;
      }
      ...
      else if ( 65 < age ) {
         rate = 575.00;
      }
   }
   else if ( maritalStatus == MaritalStatus.Married )
   ...
}
```

الإصدار المختصر من البنيان المنطقي ينبغي أن يكون كافياً ليعطيك فكرة عن مدى التعقيد الذي يمكن أن يصل إليه هذا النوع من الأشياء. إنه لا يعرض الإناث المتزوجات أو أي ذكور أو معظم الأعمار بين 18 و65. تستطيع أن تتخيل مدى التعقيد الذي يصل إليه عندما تبرمج كل جدول المعدلات.

بإمكانك القول، "حسناً، لكن لمَ تقوم باختبار كل الأعمار؟ لمَ لا تضع المعدلات في مصفوفات تبعاً لكل عمر؟" هذا سؤال جيد، أن تضع المعدلات في مصفوفات منفصلة تبعاً لكل عمر هو تحسينة واحدة واضحة.

حل أفضل، على أي حال، أن تضع المعدلات في مصفوفات تبعاً لكل العوامل، ليس العمر فقط. إليك كيف تصرح عن المصفوفة في فيجوال بيسيك:

مثال فيجوال بيسك عن التصريح عن البيانات لتهيئة جدول معدلات الضمان

```
Public Enum SmokingStatus
   SmokingStatus_First = 0
   SmokingStatus_Smoking = 0
   SmokingStatus_NonSmoking = 1
   SmokingStatus_Last = 1
End Enum
Public Enum Gender
   Gender_First = 0
   Gender_Male = 0
   Gender_Female = 1
   Gender_Last = 1
End Enum
Public Enum MaritalStatus
   MaritalStatus_First = 0
   MaritalStatus_Single = 0
   MaritalStatus_Married = 1
   MaritalStatus_Last = 1
End Enum
Const MAX_AGE As Integer = 125
Dim rateTable ( SmokingStatus_Last, Gender_Last,
MaritalStatus_Last, _MAX_AGE ) As Double
```

حالما صرحت عن المصفوفة، عليك أن تجد طريقة ما لتضع البيانات فيها.[1] تستطيع استخدام عبارات الإسناد أو قراءة البيانات من ملف من أحد الأقراص أو أن تحسب البيانات أو أن تفعل أي شيء مناسب. بعد أن تنتهي من تهيئة البيانات، تكون قد حصلت على مصفوفة جاهزة لتلبيك عندما تريد أن تحسب معدلاً. المنطق المعقد الذي رأيته سابقاً استبدل بعبارة بسيطة مثل هذه:

مثال فيجوال بيسك عن طريقة متقنة لتحديد معدل ضمان

```
rate = rateTable( smokingStatus, gender,
     maritalStatus, age )
```

هذا النهج يمتلك الحسنات العامة الناتجة عن استبدال المنطق المعقد بجدول بحث. البحث في جدول أكثر سهولة للقراءة والتغيير.

مثال تنسيق الرسالة المرن

تستطيع أن تستخدم جدولاً لتصف منطقاً مرناً جداً على أن يُمثَّل بالشفرة. في مثال تصنيف المحارف، ومثال الأيام في الشهر، ومثال معدلات الضمان، كنت على الأقل تعلم أنك تستطيع أن تكتب سلسلة طويلة من عبارات

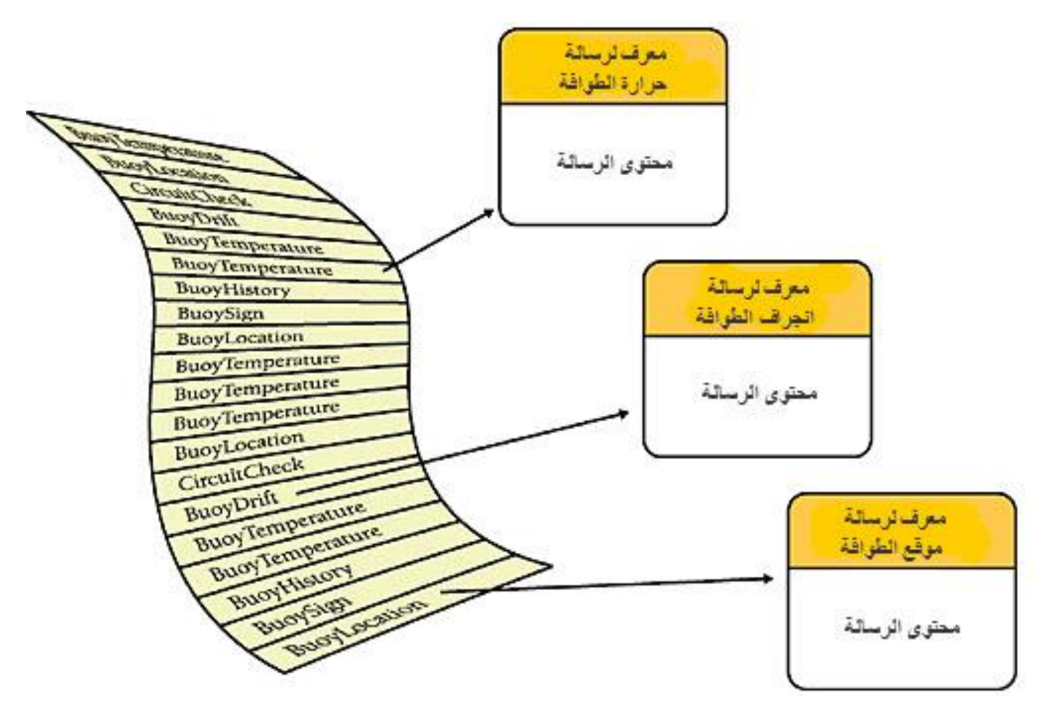
[1] **إشارة مرجعية** إحدى حسنات النهج جدولي القيادة أنك تستطيع أن تضع بيانات الجدول في ملف وتقرأها اثناء وقت التنفيذ. هذا يسمح لك أن تغير شيء ما مثل جدول معدلات الضمان دون تغير البرنامج نفسه. لمزيد حول هذه الفكرة، انظر القسم 10. 6، "وقت الربط".

if إذا احتجت لذلك. في بعض الحالات على أية حال، تكون المعلومات معقدة جداً على أن توصّف في عبارات if مشفرة بالتشفير الصعب.[1]

إذا كنت تعتقد أنك عرفت كيفية عمل جداول الدخول المباشر، قد تريد أن تتجاوز المثال التالي. إنه أكثر تعقيداً بقليل من الأمثلة السابقة، مع ذلك، يشرح بشكل أكبر قوة النُهج جدولية القيادة.

افترض أنك تكتب إجرائية لتطبع رسائل مخزنة في ملف. الملف يحوي عادة 500 رسالة، وكل ملف فيه حوالي 20 نوع من الرسائل. تأتي الرسائل بالأصل من طوافة وتعطي حرارة الماء ومكان الطوافة وما إلى ذلك.

كل واحدة من الرسائل تمتلك عدة حقول، وكل واحدة من الرسائل تبدأ بترويسة تحوي معرفاً يتيح لك معرفة أي نوع من العشرين أنت تتعامل معه. الشكل 18-2 يوضح كيف تُخزَّن الرسائل.



الشكل 18-2 لا تُخزَّن الرسائل بترتيب محدد، وكل واحدة معرَّفة بمعرف الرسالة.

تنسيق الرسائل متقلب، ومحدد من قبل الزبون، وأنت لا تمتلك السيطرة الكافية على الزبون لتجعله يوازنها. يظهر الشكل 18-3 كيف تبدو بضعة رسائل بالتفصيل.

[1] معاني التشفير الصعب يُطلق على قيم البيانات أو السلوك المكتوبة مباشرة داخل البرنامج، ربما في مناطق متعددة. والذي يتطلب دقة عند تغيير هذه "الشفرة الصعبة" بحيث لا يسبب فشل في البرنامج.

معرف لرسالة موقع الطوافة
خط العرض (رقم حقيقي)
خط الطول (رقم حقيقي)
العمق (رقم صحيح)
وقت القياس (وقت من اليوم)

معرف لرسالة انجراف الطوافة
التغير في خط العرض (رقم حقيقي)
التغير في خط الطول (رقم حقيقي)
وقت القياس (وقت من اليوم)

معرف لرسالة حرارة الطوافة
متوسط الحرارة (رقم حقيقي)
مجال الحرارة (رقم حقيقي)
عدد العينات (رقم صحيح)
الموقع (سلسلة محرفية)
وقت القياس (وقت من اليوم)

الشكل 18-3 بجانب معرف الرسالة، كل نوع من الرسائل له تنسيقه الخاص

النهج المعتمد على المنطق

إذا استخدمت النهج المعتمد على المنطق، قد تقرأ كل رسالة، وتتفحص المعرف، وبعدها تستدعي إجرائية مصممة لقراءة وتفسير وطباعة كل نوع من الرسائل. إذا كان لديك 20 نوعاً من الرسائل، سيكون لديك 20 إجرائية. وسيكون لديك أيضاً "من يعلم كم عدد" الإجرائيات هي المستوى الأدنى الضرورية لدعم الإجرائيات العشرين-مثالاً، سيكون لديك الإجرائية PrintBuoyTemperatureMessage() لطباعة رسالة حرارة الطوافة. لن يكون النهج غرضي التوجه أفضل بكثير. ستقوم بشكل اعتيادي باستخدام كائن رسالة مجرد مع صفوف فرعية لكل نوع من أنواع الرسائل.

في كل مرة يتغير فيها تنسيق أي رسالة، سيكون عليك أن تغير المنطق في الإجرائية أو الصف المسؤول عن تلك الرسالة. في الرسالة التفصيلية السابقة، إذا تغير حقل درجة الحرارة المتوسطة من فاصلة عائمة إلى شيء ما آخر، سيكون عليك أن تغير المنطق في PrintBuoyTemperatureMessage(). (إذا تغيرت الطوافة نفسها من "فاصلة عائمة" إلى شيء ما آخر، سيكون عليك أن تحصل على طوافة جديدة!)

في النهج المعتمد على المنطق، تتألف إجرائية قراءة الرسالة من حلقة لقراءة كل رسالة، واستخراج المعرف، وبعدها استدعاء واحدة من الإجرائيات العشرين اعتماداً على معرف الرسالة. إليك الشفرة الزائفة للنهج المعتمد على المنطق[1]:

```
While more messages to read
 Read a message header
```

[1] **إشارة مرجعية** تستخدم هذه الشفرة الزائفة منخفضة المستوى لغرض مختلف عن الشفرة الزائفة التي تستخدمها لتصميم الإجرائية. لتفاصيل حول التصميم بالشفرة الزائفة، انظر الفصل 9، "عملية برمجة الشفرة الزائفة"

```
Decode the message ID from the message header
If the message header is type 1 then
Print a type 1 message
Else if the message header is type 2 then
Print a type 2 message
...
Else if the message header is type 19 then
Print a type 19 message
Else if the message header is type 20 then
Print a type 20 message
```

تم اختصار الشفرة الزائفة لأنك تستطيع أن ترى الفكرة دون رؤية الحالات العشرين كلها.

النهج غرضي التوجه

إذا كنت تستخدم نهجاً غرضي التوجه جامداً، سيُخبّئ المنطق في بنيان وراثة الكائنات لكن البنيان الأساسي سيكون تماماً بنفس التعقيد:

```
While more messages to read
 Read a message header
 Decode the message ID from the message header
 If the message header is type 1 then
 Instantiate a type 1 message object
 Else if the message header is type 2 then
 Instantiate a type 2 message object
 ...
 Else if the message header is type 19 then
 Instantiate a type 19 message object
 Else if the message header is type 20 then
 Instantiate a type 20 message object
 End if
End While
```

بغض النظر عن كون المنطق مكتوباً مباشرة أو محتوً في صفوف خاصة، سيكون لكل من الأنواع العشرين للرسائل إجرائيتها الخاصة لطباعة الرسالة الموافقة. يمكن أن تُمثل كل إجرائية بالشفرة الزائفة. هذه هي الشفرة الزائفة لإجرائية تقرأ وتطبع رسالة حرارة الطوافة:

```
Print "Buoy Temperature Message"

Read a floating-point value
Print "Average Temperature"
Print the floating-point value
```

```
Read a floating-point value
Print "Temperature Range"
Print the floating-point value

Read an integer value
Print "Number of Samples"
Print the integer value

Read a character string
Print "Location"
Print the character string

Read a time of day
Print "Time of Measurement"
Print the time of day
```

هذه هي الشفرة لواحد فقط من أنواع الرسائل. كل من الأنواع التسعة عشر الباقية تتطلب شفرة مماثلة. وإذا أضيف النوع الحادي والعشرين، سيكون ضرورياً أن يضاف إما الإجرائية الواحدة والعشرين أو الصف الجزئي الواحد والعشرين-في كلتا الحالتين سيتطلب النوع الجديد أن تتغير الشفرة.

النهج جدولي القيادة

النهج جدولي القيادي اقتصادي أكثر من النهج السابق. إجرائية قراءة الرسالة تتألف من حلقة تقرأ ترويسة كل رسالة، فتستخرج المعرف، فتبحث عن توصيف الرسالة في المصفوفة Message، ثم تستدعي الإجرائية ذاتها كل مرة لتفك شفرة الرسالة. بالنهج جدولي القيادة، تستطيع أن تصف تنسيق كل رسالة من الرسائل في جدول بدلاً من التشفير الصعب في منطق البرنامج. هذا يجعل كتابة الشفرة أسهل أصلاً، ويولد شفرة أقل، ويجعل الصيانة أسهل بدون الحاجة لتغيير الشفرة.

لاستخدام هذا النهج، عليك أن تبدأ بذكر أنواع الرسائل وأنواع الحقول. في سي++، تستطيع تحديد أنواع كل الحقول المحتملة بهذه الطريقة:

مثال بلغة سي++ عن تحديد أنواع بيانات الرسالة

```
enum FieldType {
    FieldType_FloatingPoint,
    FieldType_Integer,
    FieldType_String,
    FieldType_TimeOfDay,
    FieldType_Boolean,
    FieldType_BitField,
    FieldType_Last = FieldType_BitField
};
```

بدلاً من التشفير الصعب لإجرائيات طباعة لكل من الأنواع العشرين للرسائل، تستطيع إنشاء بضعة إجرائيات تطبع كل من الأنواع الرئيسية للبيانات-فاصلة عائمة (حقيقي)، صحيح، سلسلة محرفية... تستطيع توصيف محتوى كل نوع من الرسائل في جدول (متضمناً اسم ذلك الحقل) ثم تفك شفرة كل رسالة بالاعتماد على التوصيف في الجدول. قد يبدو مدخل إلى جدول يصف نوع واحد من الرسائل كهذا:

مثال عن تعريف مدخل في جدول الرسائل

```
Message Begin
    NumFields 5
    MessageName "Buoy Temperature Message"
    Field 1, FloatingPoint, "Average Temperature"
    Field 2, FloatingPoint, "Temperature Range"
    Field 3, Integer, "Number of Samples"
    Field 4, String, "Location"
    Field 5, TimeOfDay, "Time of Measurement"
Message End
```

يمكن أن يُشفر تشفيراً صعباً في البرنامج (في هذه الحالة، كل العناصر الظاهرة ستسند إلى متحولات)، أو قد تُقرأ من ملف عند إقلاع البرنامج أو بعد.

حالما تتم قراءة تعاريف الرسائل إلى البرنامج، بدلاً من كون كل المعلومات مُضمنة في منطق البرنامج، تكون قد حصلت على هذه المعلومات مضمنةً في بيانات. تميل البيانات لتكون أكثر مرونة من المنطق. البيانات سهلة التغيير عندما يتغير تنسيق الرسالة. إذا كان عليك أن تضيف نوعاً جديداً من الرسائل، فإنك تستطيع ببساطة إضافة عنصر آخر إلى جدول البيانات.

هذه الشفرة الزائفة للحلقة في المستوى الأعلى في النهج جدولي القيادة:

```
While more messages to read
 Read a message header
 Decode the message ID from the message header
 Look up the message description in the message-description table
 Read the message fields and print them based on the message description
End While
```

الأسطر الثلاثة الأولى هي نفسها في النهج المعتمد على المنطق.

على خلاف الشفرة الزائفة التابعة للنهج المعتمد على المنطق، لم تختصر الشفرة في هذه الحالة لأن المنطق أقل تعقيد بكثير. في المنطق تحت هذا المستوى، ستجد إجرائية واحدة فقط قادرة على تفسير توصيف الرسالة من جدول توصيف الرسائل، وقراءة معلومات الرسالة، وطباعة الرسالة. هذه الإجرائية أعم من أي إجرائية لطباعة الرسائل في النهج المعتمد على المنطق لكنها ليست معقدة أكثر بكثير، وستكون إجرائية واحدة بدلاً من 20.

```
While more fields to print
 Get the field type from the message description
 case ( field type )
 of ( floating point )
  read a floating-point value
  print the field label
  print the floating-point value
```

```
 of ( integer )
  read an integer value
  print the field label
  print the integer value

 of ( character string )
  read a character string
  print the field label
  print the character string

 of ( time of day )
  read a time of day
  print the field label
  print the time of day

 of ( boolean )
  read a single flag
  print the field label
  print the single flag

 of ( bit field )
  read a bit field
  print the field label
  print the bit field
 End Case
End While
```

بلا إنكار، هذه الإجرائية ذات الحالات الست أطول من الإجرائية المفردة التي احتجناها لطباعة رسالة حرارة الطوافة. لكن هذه هي الإجرائية الوحيدة التي ستحتاج. لن تحتاج 19 إجرائية أخرى ل 19 نوع آخر من الرسائل. هذه الرسالة تتعامل مع أنواع الحقول الستة وتهتم بشأن أنواع الرسائل كلها.

هذه الإجرائية تظهر الطريقة الأعقد لتحقيق هذا النوع من البحث في الجدول لأنها تستخدم عبارة case. نهجٌ آخر، أن تنشئ الصف المجرد AbstractField وبعدها تنشئ صفوف فرعية لكل نوع من أنواع الحقول. لن تحتاج عبارة case؛ تستطيع أن تستدعي الإجرائية العضو للنوع المناسب من الكائنات.

إليك كيف تهيئ أنواع الكائنات في سي++:

```
مثال بلغة سي++ عن تهيئة أنواع الكائنات
class AbstractField {
    public:
    virtual void ReadAndPrint( string, FileStatus & )
                  =0;
};
class FloatingPointField : public AbstractField {
    public:
    virtual void ReadAndPrint( string, FileStatus & ) {
    ...
    }
};
class IntegerField ...
class StringField ...
...
```

مقطع الشفرة هذا يوضح الإجرائية العضو في كل صف التي لديها وسيط من نوع سلسلة نصية ووسيط FileStatus.

الخطوة التالية أن نصرح عن مصفوفة لنمسك مجموعة الكائنات. المصفوفة هي جدول البحث، وإليك كيف تبدو:

```
مثال بلغة سي++ عن تهيئة جدول لإمساك كائن من كل نوع
AbstractField* field[ Field_Last+1];
```

الخطوة الأخيرة المطلوبة لتهيئة جدول الكائنات هي أن تسند أسماء الكائنات الخاصة إلى المصفوفة Field:

```
مثال بلغة سي++ عن تهيئة لائحة من الكائنات
field[ Field_FloatingPoint ] = new FloatingPointField();
field[ Field_Integer ] = new IntegerField();
field[ Field_String ] = new StringField();
field[ Field_TimeOfDay ] = new TimeOfDayField();
field[ Field_Boolean ] = new BooleanField();
field[ Field_BitField ] = new BitFieldField();
```

مقطع الشفرة هذا يفترض أن FloatingPointField والمعرّفات الأخرى في الجهة اليمنى من عبارات الإسناد هي أسماء كائنات من النوع AbstractField. إسناد الكائنات إلى العناصر في المصفوفة يعني أنك تستطيع أن تستدعي الإجرائية ReadAndPrint() الصحيحة بالرجوع إلى عنصر المصفوفة بدلاً من استخدام أنواع خاصة من الكائنات بشكل مباشر.

حالما تتم تهيئة جدول الإجرائيات، تستطيع أن تتعامل مع حقل في الرسالة ببساطة عن طريق الدخول إلى جدول الكائنات واستدعاء واحدة من الإجرائيات الأعضاء في الجدول. الشفرة تبدو كالتالية:

مثال بلغة سي++ عن البحث عن الكائنات والأعضاء

```
fieldIdx = 1;
while ( ( fieldIdx <= numFieldsInMessage ) && ( fileStatus == OK
) ) {
    fieldType = fieldDescription[ fieldIdx
      ].FieldType;
    fieldName = fieldDescription[ fieldIdx
      ].FieldName;
    field[ fieldType ].ReadAndPrint(
      fieldName, fileStatus );
    fieldIdx++;
}
```

هذا البحث في الجدول الذي يستدعي الإجرائية بالاعتماد على نوع الحقل-فقط بالبحث عنه في جدول الكائنات

تذكر شفرة البحث في الجدول الأصلية التي كانت تحتوي على 34 سطر متضمنة عبارة case؟ إذا استبدلت عبارة case بجدول كائنات، هذا سيكون كل الشفرة التي ستحتاج لتؤمن نفس الآلية. بشكل غير معقول، إنه أيضاً كل الشفرة التي ستحتاج لاستبدال كل الإجرائيات العشرين المستقلة في النهج المعتمد على المنطق، أكثر بعد، إذا كانت توصيفات الرسالة تُقرأ من ملف، أنواع الرسائل الجديدة لن تتطلب تغيرات في الشفرة مالم يكن هناك أنواع حقول جديدة.

تستطيع استخدام هذا النهج في أي لغة غرضية التوجه. إنه أقل وهناً أمام الأخطاء، وأكثر سهولة في الصيانة، وأكثر فاعلية من عبارات if الطويلة، أو عبارات case أو الصفوف الكثيرة.

حقيقة كون التصميم يستخدم الوراثة وتعدد الأشكال لا يجعله تصميماً جيداً. التصميم غرضي التوجه الجامد المشروح سابقاً في قسم "النهج غرضي التوجه" سيتطلب شفرة بقدر التصميم الإجرائي الجامد-أو أكثر. ذلك التصميم يجعل فضاء الحل أكثر تعقيداً، بدلاً من أقل. الرؤية التصميمية المفتاحية في هذه الحالة هي ليست غرضية التوجه ولا إجرائية التوجه-إنها تستخدم الفكر النير المتعلق بجداول البحث.

مراوغة مفاتيح البحث

في الأمثلة الثلاثة السابقة، تستطيع استخدام المعلومات لتفتح مداخل الجدول مباشرة. هذا يعني، إنك تستطيع أن تستخدم messageID كمفتاح دون أي تعديل، كما تستطيع استخدام month في مثال الأيام في الشهر، وgender & maritalStatus & smokingStatus في مثال معدلات الضمان.

سترغب دائما بالدخول إلى الجدول مباشرة لأنه بسيط وسريع. أحياناً، على أية حال، لا تكون المعلومات متعاونة. في مثال معدلات الضمان، لم يحسن العمر التصرف. المنطق الأصلي كان لديه معدل واحد للناس تحت 18، ومعدلات مستقلة للأعمار من 18 إلى 65، ومعدل واحد للناس فوق 65. هذا عنى أنه من الأعمار من 0 إلى 17 ومن 66 حتى النهاية، لا تستطيع أن تستخدم العمر للدخول مباشرة إلى جدول يحوي فقط مجموعة واحدة من المعدلات لعدة أعمار.

هذا يقود إلى موضوع مراوغة مفاتيح جداول البحث. تستطيع أن تراوغ على المفاتيح بعدة طرق:

كرر المعلومات لتجعل المفتاح يعمل بشكل مباشر طريقة مستقيمة لجعل العمر يعمل كمفتاح إلى جدول المعدلات أن تكرر معدلات تحت 18 لكل الأعمار من 0 إلى 17 وبعدها تستخدم العمل كمفتاح مباشر إلى الجدول. تستطيع أن تفعل نفس الشيء للأعمار من 66 إلى النهاية. فوائد هذا النهج هي أن بنيان الجدول نفسه مستقيم والدخول إلى الجدول مستقيم أيضاً. إذا احتجت أن تضيف معدلات خاصة بالعمر للأعمار من 17 ونزولاً، تستطيع أن تغير الجدول فقط. السلبيات هي أن التكرار سيضيع مساحة من أجل المعلومات الزائدة ويزيد احتمالية الأخطاء في الجدول-أتمنى أن تكون فقط لأن الجدول يحتوي معلومات زائدة.

حول المفتاح كي تجعله يعمل بشكل مباشر طريقة ثانية لجعل العمر يعمل كمفتاح مباشر هي تطبيق المجال المحدد ذو سلوك حسن كفاية بحيث تستطيع أن تستخدم التابعين min() & max() لتصنع التحويل. على سبيل المثال، تستطيع أن تستخدم التعبير

```
max( min( 66،Age )17، )
```

لإنشاء مفتاح جدول يتغير من 17 إلى 66.

إنشاء تابع التحويل يتطلب منك التعرف على نموذج في المعلومات لتستخدمه كمفتاح، وهذا ليس دائماً بسيط بقدر استخدام min() & max(). افترض أنه في هذا المثال قد كانت المعدلات لمجموعات عمرية لكل خمس سنوات بدلاً من مجموعات السنة الواحدة. مالم ترد تكرار كل بياناتك خمس مرات، سيكون عليك أن تأتي بتابع يقسم العمر على 5 بشكل مناسب ويستخدم الإجرائيتين min() & max().

اعزل تحويل المفتاح في إجرائيته الخاصة إذا كان عليك أن تراوغ البيانات لتجعلها تعمل كمفتاح جدول، ضع العملية التي تحول البيانات إلى مفتاح في إجرائيتها الخاصة. تلغي الإجرائية احتمالية استخدام تحويلات مختلفة في مناطق مختلفة. إنها تجعل التعديل أسهل عندما يتغير التحويل. اسمٌ جيد للإجرائية، مثل KeyFromAge()، أيضاً يوضح ويوثق غرض الآلية الرياضية.

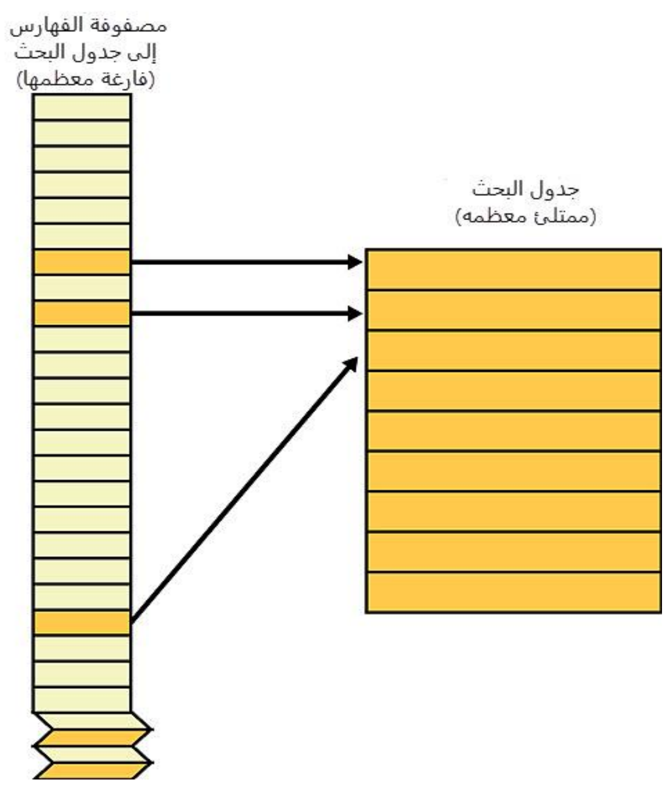
إذا كانت بيئتك تؤمن تحويلات مفتاح "مسبقة الصنع"، استخدمها. مثالاً، جافا تؤمن HashMap، والذي يمكن أن يستخدم لربط أزواج مفتاح/قيمة.

3 .18 جداول الدخول المفهرس

في بعض الأحيان لا يكون التحويل الرياضي البسيط قوياً كفاية ليصنع القفزة من معلومة مثل العمر إلى مفتاح الجدول. هكذا حالات ملائمة لاستخدام مخطط الدخول المفهرس.

عندما تستخدم الفهارس، فإنك تستخدم المعلومة الرئيسية لتبحث عن مفتاح في جدول الفهرس، فتستخدم القيمة من جدول الفهرس لتبحث عن المعلومة الرئيسية التي تهمك.

لنفرض أنك تدير مخزن ولديك مستودع يحوي حوالي 100 مادة. لنفرض بعد، أن كل مادة لها رقم تفريق بأربع خانات والذي يصنع مجالاً من 0000 إلى 9999. في هذه الحالة، إذا أردت استخدام رقم التفريق لتفتح وبشكل مباشر الجدول الذي يذكر بعض المعلومات عن كل مادة، ستهيئ مصفوفة فهارس ب 10000 مدخل (من 0 إلى 9999). المصفوفة فارغة ما عدا المئة مدخل التي توافق أرقام التفريق للمئة مادة الموجودة في المخزن. كما يظهر الشكل 18-4، هذه المداخل تشير إلى جدول توصيف المواد الذي يحوي أقل بكثير من 10000 مدخل.



الشكل 18-4 بدلاً من الدخول المباشر، يتم الدخول إلى جدول الدخول المفهرس عبر فهرس وسيط.

مخططات الدخول المفهرس تقدم حسنتين رئيسيتين. أولاً، إذا كانت كل المداخل في جدول البحث الرئيسي ضخمة، إنشاء مصفوفة فهارس مع الكثير من المساحة الضائعة سيأخذ مساحة أقل بكثير مما يأخذ إنشاء جدول بحث رئيسي مع الكثير من المساحة الضائعة. مثالاً، افترض أن الجدول الرئيسي يستهلك 100 بايت لكل مدخل وأن مصفوفة الفهارس تأخذ بايتين لكل مدخل. افترض أن الجدول الرئيسي يحوي 100 مدخل وأن المعلومة المستخدمة للدخول إليه لها 10000 قيمة محتملة. في هذه الحالة، الخيار بين أن يكون لديك فهرس ب 10000 مدخل أو جدول بيانات رئيسي ب 10000 مدخل. إذا استخدمت الفهرس، فإن المساحة الكلية التي تستهلكها 30000 بايت. إذا امتنعت عن البنيان المفهرس وضيعت المساحة في الجدول الرئيسي، سيكون استهلاكك الكلي للمساحة 1000000 بايت.

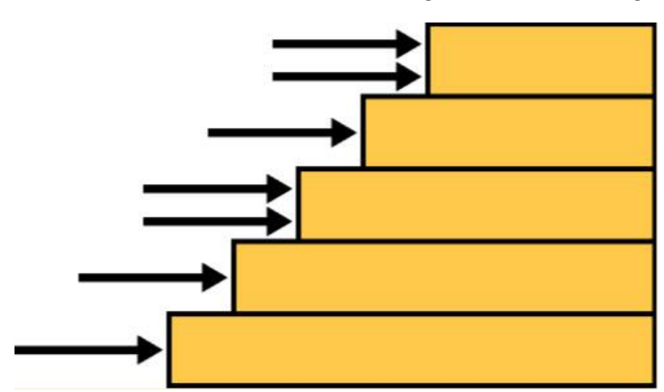
الحسنة الثانية، حتى لو لم توفر مساحة باستخدام الفهرس، أنه في بعض الأحيان تكون معالجة مداخل الفهرس أقل كلفة منها في مداخل الجدول الرئيسي. مثالاً، إذا كان لديك جدول يحوي أسماء موظفين وتواريخ توظيفهم ورواتبهم، تستطيع أن تنشئ فهرساً لتدخل إلى الجدول عبر اسم الموظف، وآخراً لتدخل إليه عبر تاريخ التوظيف، وثالثاً يدخل إلى الجدول عبر الراتب.

حسنة أخيرة لمخطط الدخول المفهرس هي الحسنة العامة لجدول البحث في إمكانية الصيانة. البيانات المشفرة في جدول أسهل صيانةً من البيانات المضمنة في الشفرة. كي تعظم المرونة، ضع شفرة الدخول المفهرس في إجرائيتها الخاصة واستدعي الإجرائية عندما تحتاج مفتاح للجدول من رقم التفريق. عندما يحين وقت تغيير الجدول، قد تقرر أن تبدل مخطط الدخول المفهرس أو تبدل إلى مخطط "بحث مجدول" آخر كلياً. مخطط الدخول سيكون أسهل للتغيير إذا لم تبعثر مرات الدخول إلى الفهرس في برنامجك.

18.4 جداول الدخول بدرجة الدرج

نوع آخر بعد للدخول إلى الجدول هو طريقة درجة الدرج. طريقة الدخول هذه ليست مباشرة مثل بنيان الفهرس، لكنها لا تضيع مساحة تخزين مثلها.

الفكرة الرئيسية من بنيان درجة الدرج، موضحة بالشكل 18-5، وهي أن المدخلات في الجدول هي صالحة لمجالات من المعلومات وليس لنقاط محددة من المعلومات.



الشكل 18-5 نهج درجة الدرج يصنف كل مدخل بتحديد مستوى اصطدامه ب "الدرج." "الدرجة" التي يصطدم بها تحدد فئته

على سبيل المثال، إذا كنت تكتب برنامج تقييم، المدخل "B" يتراوح بين 75 بالمئة إلى 90 بالمئة. إليك مجال الدرجات التي قد يتوجب عليك أن تبرمجها يوماً ما.

≥90.0% A
< 90.0% B
< 75.0% C
< 65.0% D

< 50.0%   F

هذا نطاق مزعج لجدول بحث لأنك لا تستطيع استخدام تحويل معلومات بسيط لتفتح على الحروف من A إلى F. مخطط الفهرس لن يكون ملائماً لأن الأرقام تستخدم نظام الفاصلة العائمة (حقيقية). قد تفكر في تحويل أرقام الفاصلة العائمة إلى أرقام صحيحة، وبهذه الحالة سيكون خيار تصميم صالح، لكن لمصلحة الشرح، هذا المثال سيلتزم بالفاصلة العائمة.

لاستخدام طريقة درجة الدرج، نقوم بوضع النهاية العليا لكل مجال في جدول وبعدها نكتب حلقة لمقارنة النتيجة بكل نهاية عليا من النهايات (بدءاً من أسفل الدرج). عندما نصل إلى أول توقف للنتيجة عند أعلى مجال، نكون قد عرفنا الدرجة (العلامة). مع تقنية درجة الدرج، يتوجب عليك أن تكون حذراً حتى تتعامل مع نقاط النهاية للمجالات بطريقة صحيحة. إليك الشفرة بلغة فيجوال بيسك التي تسند العلامات لمجموعة من الطلاب بالاعتماد على هذا المثال:

مثال بلغة فيجوال بيسك للبحث في جدول دخول بدرجة الدرج

```
' set up data for grading table
Dim rangeLimit() As Double = { 50.0, 65.0, 75.0,
                               90.0, 100.0 }
Dim grade() As String = { "F", "D", "C", "B", "A" }
maxGradeLevel = grade.Length - 1
...
' assign a grade to a student based on the student's score
gradeLevel = 0
studentGrade = "A"
While ( ( studentGrade = "A" ) and
( gradeLevel < maxGradeLevel ) )
If ( studentScore < rangeLimit( gradeLevel ) ) Then
studentGrade = grade( gradeLevel )
End If
gradeLevel = gradeLevel + 1
Wend
```

على الرغم من أن هذا مثال بسيط، تستطيع بسهولة تعميمه لتتعامل مع عدة تلاميذ وعدة مخططات للتقييم (على سبيل المثال، درجات مختلفة لمستويات مختلفة بالنسبة لوظائف (مهام) مختلفة)، وتتعامل أيضاً مع التغيرات في مخطط التقييم.

حسنة هذا النهج على الطرق جدولية القيادة الأخرى هي أنه يعمل بنجاح مع المعلومات غير المنتظمة. مثال التقييم بسيط في ذلك، على الرغم من أن الدرجات يتم تعيينها بفواصل غير منتظمة، والأرقام "مقرّبة،" تنتهي بخمسات وأصفار. نهج درجة الدرج يناسب بشكل تام وجيد المعلومات التي لا تنتهي بصورة منتظمة بخمسات وأصفار. تستطيع استعمال نهج درجة الدرج في أعمال الإحصاء لاحتمالات التوزيع مع الأرقام مثل هذه:

الاحتمال	مقدار مطالبة التأمين
0.458747	$0.00
0.547651	$254.32
0.627764	$514.77
0.776883	$747.82
0.893211	042.65،$1

0.957665	887.55،$5
0.976544	836.98،$12
0.987889	234.12،$27

...

الأرقام البشعة مثل هذه ترفض أي محاولة لنحصل على تابع يحولها بشكل منتظم إلى مفاتيح جدول. نهج درجة الدرج هو الحل.

يتمتع هذا النهج بالحسنات العامة للنُهج جدولية القيادة: إنه مرن وسهل التعديل. إذا كانت مجالات التقييم في مثال التقييم عرضة للتغير، يمكن أن يُكيّف البرنامج بسهولة عن طريق تعديل المداخل في المصفوفة `RangeLimit`. تستطيع بسهولة أن تعمم القسم المتخصص بتعيين الدرجات من البرنامج بحيث يقبل جدول درجات وقيم الفصل الموافقة. لا ينبغي للقسم المتخصص بتعيين الدرجات من البرنامج أن يستخدم قيم الفصل باعتبارها نسب مئوية؛ إنه يستطيع أن يستخدم نقاط بعينها بدلاً من النسب المئوية، ولا ينبغي أن يتغير البرنامج كثيراً.

إليك بعض الأمور الدقيقة لتضعها في حسبانك عندما تستخدم تقنية درجة الدرج:

راقب نقاط النهايات تأكد من أنك غطيت حالة النهاية العليا من كل مجال درجة درج. شغّل بحث درجة الدرج بحيث يجد عناصر توافق أي مجال غير المجال الأعلى، وبعدها دع البقية تسقط في المجال الأعلى. في بعض الأحيان هذا يتطلب قيمة اصطناعية لقمة المجال الأعلى.

كن حذراً من عدم التفريق بين <(أصغر) و<=. تأكد أن الحلقة تنتهي بشكل صحيح بقيم تسقط إلى المجالات الأعلى وأن حواف المجال تم التعامل معها بشكل صحيح.

ضع بحسبانك استخدام البحث الثنائي بدلاً من البحث التعاقبي في مثال التقييم، تبحث الحلقة التي تعيين الدرجات بشكل تعاقبي عبر لائحة نهايات التقييم. إذا كان لديك لائحة أضخم، كلفة البحث التعاقبي قد تصبح غير قابلة للدفع. إذا كان ولا بد، تستطيع أن تستبدلها ببحث شبيه بالبحث الثنائي. إنه "شبيه" بالبحث الثنائي لأن غاية معظم البحوث الثنائية أن تجد قيمة. في هذه الحالة، إنك لا تتوقع أن تجد قيمة؛ بل الفئة الصحيحة للقيمة. يجب حتماً على خوارزمية البحث الثنائي أن تحدد وبصحة أين ينبغي أن تذهب القيمة. تأكد أيضاً من التعامل مع نقطة النهاية كحالة خاصة.

ضع بحسبانك استخدام الدخول المفهرس بدلاً من تقنية درجة الدرج قد يكون مخطط الدخول المفهرس كالذي وصف في القسم 18. 3 بديل جيد لتقنية درجة الدرج. البحث المطلوب في طريقة درجة الدرج يمكن ان يعطي معنى، وإذا كانت سرعة التنفيذ أمراً هاماً، قد تكون راغباً بشراء المساحة التي يأخذها بنيان الفهرس الإضافي مقابل ميزة الزمن التي ستحصل عليها بطريقة أكثر مباشرة بالدخول.

كما هو واضح، هذا البديل ليس خياراً جيداً في كل الحالات. في مثال التقييم، قد تستطيع استخدامه؛ إذا كان لديك فقط 100 نقطة مئوية مستقلة، كلفة الذاكرة المطلوبة لتهيئة مصفوفة فهارس لن تكون غير قابلة للدفع. إذا، من الناحية الأخرى، كان لديك معلومات الاحتمالات المذكورة سابقاً، لن تستطيع تهيئة مخطط الفهرسة لأنك لن تستطيع أن تفتح على مداخل بأرقام مثل 0.458747 و0.547651.

في بعض الحالات، أي من الخيارات المتعددة قد يعمل[1]. الهدف من التصميم هو اختيار واحد من الخيارات الجيدة المتعددة لحالتك. لا تقلق كثيراً جداً بشأن اختيار أفضل واحد. كما يقول باتلر لامبسن، مهندس معروف في مايكروسوفت، من الأفضل أن تكافح للحصول على حل جيد وتتجنب كارثة بدلاً من محاولة إيجاد الحل الأفضل (لامبسن 1984).

ضع البحث بجدول درجة الدرج في إجرائيته الخاصة عندما تنشئ آلية تحويل تبدل قيمة مثل `StudentGrade` إلى مفتاح جدول، ضع هذه الآلية في إجرائيتها الخاصة.

18. 5 أمثلة أخرى على جداول البحث

تظهر أمثلة أخرى على جداول البحث في أقسام أخرى من الكتاب. لقد استخدمت في مسار مناقشة التقنيات الأخرى، ولم يركز السياق على جداول البحث ذاتها. إليك أين تجدها:

- البحث على المعدلات في جدول الضمان: القسم 16. 3، "إنشاء حلقات بسهولة من الداخل إلى الخارج"
- استخدام جداول القرار لتحل محل المنطق المعقد: "استخدم جداول القرار لتحل محل الشروط المعقدة" في القسم 19. 1
- كلفة تصفح الذاكرة خلال بحث في جدول: القسم 25. 3، "أنواع السمن والدبس"
- تجميعات لقيم منطقية (أ أو ب أو ج): "بدل جداول البحث مكان التعابير المعقدة" في القسم 26. 1
- قيم محسوبة مسبقاً في جدول تسديد قرض: القسم 26. 4، "التعابير."

1 **إشارة مرجعية** للمزيد حول النهج الجيدة لاختيار بدائل تصميمية، انظر الفصل 5، "التصميم في البناء".

لائحة مهام: الطرق جدولية القيادة1

- هل وضعت في حسبانك الطرق جدولية القيادة كبديل للمنطق المعقد؟
- هل وضعت في حسبانك الطرق جدولية القيادة كبديل لبنيان الوراثة المعقد؟
- هل وضعت في حسبانك تخزين بيانات الجدول في مكان خارجي والقراءة منه خلال وقت التنفيذ بحيث يمكن أن تعدل البيانات دون تغيير الشفرة؟
- إن كان الدخول المباشر إلى الجدول غير ممكن بواسطة فهرس مصفوفة صريح. (كما في مثال العمر)، هل وضعت حسابات مفتاح الدخول في إجرائية بدلاً من تكرار حسابات الفهرس في الشفرة؟

نقاط مفتاحية

- تؤمن الجداول بديل لبنيان الوراثة والمنطق المعقد. إذا وجدت أنك محتار في منطق البرنامج أو شجرة الوراثة، اسأل نفسك إن كنت تستطيع أن تبسط باستخدام جداول البحث.
- اعتبارٌ مفتاحي في استخدام جدول هو اتخاذ قرار بكيفية الدخول إلى الجدول. تستطيع الدخول إلى جداول باستخدام الدخول المباشر، أو الدخول المفهرس، أو الدخول بدرجة الدرج.
- اعتبارٌ مفتاحي آخر في استخدام جدول هو اتخاذ قرار بما سيتم وضعه في الجدول بالضبط.

[1] cc2e.com/1872

الفصل 19

قضايا التحكم العامة

المحتويات[1]

مواضيع ذات صلة

- الشفرة الخطية: الفصل 14
- الشفرة مع الجمل الشرطيّة: الفصل 15
- الشفرة مع الحلقات: الفصل 16
- هياكل التحكم غير العادية: الفصل 17
- التعقيد في تطوير البرمجيات: " الحتمية التقنية الأساسية للبرمجية: إدارة التعقيد"، في الفصل 2.5.

لا يوجد مناقشة عن التحكم كاملة إذا لم تؤدي إلى عدة قضايا عامة، تنهض فجأة عندما تفكر ببنى التحكم. معظم المعلومات في هذا الفصل مُفصّلة وواقعيّة. إذا كنت تقرأ بهدف فهم نظرية هياكل التحكم بدلًا من هدف الحصول على التفاصيل الدقيقة، عندها خُذ بعين الاعتبار المنظور التاريخي للبرمجة الهيكلية في القسم 5.19 والعلاقات بين هياكل التحكم في القسم 6.19.

[1] cc2e.com/1465

1.19 التعابير المنطقية (البوليانية):

باستثناء هيكلية التحكم الأبسط، تلك التي تستدعي تعابير العبارات البرمجية بشكل متسلسل، فإنه تعتمد كل هياكل التحكم الباقية على التعابير المنطقية (البوليانية).

استخدام صح true أو خطأ false للاختبارات المنطقية

استخدم المعرّفات true أو false في التعابير المنطقية، بدلًا من استخدام القيم 14 و0. تملك معظم لغات البرمجة الحديثة نوع البيانات البوليانية، وتؤمن مُعرفات true وfalse. أنهم يسهلون هذا الأمر- حتى أنهم لا يسمحون لك بإسناد قيم أخرى غير true وfalse للمتغيرات المنطقية. تتطلب منك لغات البرمجة، التي لا تملك نوع البيانات البولياني، الحصول على المزيد من الانضباط لجعل التعبيرات المنطقية قابلة للقراءة. فيما يلي مثال عن هذه المشكلة:



مثال بلغة البرمجة فيجول بيسيك عن استخدام أعلام غامضة للقيم المنطقية.

```
Dim printerError As Integer
Dim reportSelected As Integer
Dim summarySelected As Integer
...
If printerError = 0 Then InitializePrinter()
If printerError = 1 Then NotifyUserOfError()
                If reportSelected = 1 Then PrintReport()
If summarySelected = 1 Then PrintSummary()
If printerError = 0 Then CleanupPrinter()
```

إذا كان استخدام أعلام مثل 0 و1 هو ممارسة عملية شائعة فما هو الغلط من هذا؟ إنه غير واضح من قراءة الشفرة فيما إذا يتم تنفيذ استدعاءات التابع عندما الاختبارات صحيحة أو عندما تكون خاطئة. لا شيء من مقطع الشفرة نفسه يخبرك فيما إذا كان العدد 1 يُمثل الصح true، والعدد 0 يُمثل الخطأ، أو فيما إذا كان العكس صحيح. حتى أنه غير واضح فيما إذا تم استخدام القيم 1 و0 لتمثيل القيم true وfalse. على سبيل المثال، في سطر العبارة البرمجية:

`If reportSelected = 1`، من الممكن للعدد 1 أن يُمثل بسهولة التقرير الاول، والعدد 2 التقرير الثاني، والعدد 3 التقرير الثالث؛ لا يوجد شيء في الشفرة يخبرك بأن العدد 1 يمثل إما true أو false. وأيضًا من السهولة كتابة 0 عندما تقصد 1 أو العكس بالعكس.

استخدم المصطلحات المُسماة true وfalse من أجل اختبارات التعابير المنطقية. إذا لم تكن تدعم لغتك البرمجية هكذا مصطلحات بشكل مباشر، أنشأها باستخدام مسجلات"ماكروات" المعالجة أو المتغيرات العامة. تمت إعادة كتابة شفرة المثال السابق هنا باستخدام قيم True وFalse المُضمنة بلغة البرمجة مايكروسوفت فيجول بيسك:

مثال جيد وليس بعظيم بلغة البرمجة فيجول بيسيك عن استخدام True وFalse للاختبارات بدلًا من استخدام القيم العدديّة. [1]

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( printerError = False ) Then InitializePrinter()
If ( printerError = True ) Then NotifyUserOfError()
If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected = True ) Then PrintSummary()
If ( printerError = False ) Then CleanupPrinter()
```

يجعل استخدام الثوابت True وFalse القصد أوضح. وليس عليك أن تتذكر ماذا تُمثل 1 وماذا تُمثل 0، ولن تعكسهم عن طريق الخطأ. علاوة على ذلك، في الشفرة المُعاد كتابتها، الآن واضح أن بعض الواحدان والأصفار في مثال الفيجزل بيسك الأصلي لم يتم استخدامها كأعلام لقيم منطقية. إن السطر البرمجي `If reportSelected = 1` لم يكن أبدًا اختبار منطقي؛ بل قام باختبار فيما إذا تم اختيار التقرير الأول.

تُخبر هذه الطريقة القارئ بأنك تقوم باختبار منطقي. كما أن كتابة true عندما تعني false، أصعب في هذه الحالة من كتابة 1 عندما تعني 0، كما أنك بهذه الطريقة تتجنّب نشر الأرقام السحرية 0 و1 على امتداد الشفرة. هنا بعض النصائح عن تعريف true وfalse في الاختبارات المنطقية:

قارن بشكل ضمني القيم المنطقية مع القيم true وfalse. يمكنك أن تكتب اختبارات أوضح عن طريق التعامل مع التعابير كتعابير منطقية. على سبيل المثال، اكتب

```
while ( not done ) ...
while ( a > b ) ...
```

بدلًا من كتابة

```
while ( done = false ) ...
while ( (a > b) = true ) ...
```

[1] **إشارة مرجعية:** للمزيد حول طريقة أفضل للقيام بهذه الاختبارات نفسها، انظر مثال الشفرة التالية.

يُخفض استخدام المقارنات الضمنية من عدد العناصر التي على شخص ما يقرأ شفرتك أن يحتفظ فيها بعقله، ويتم قراءة التعابير الناتجة بشكل أكبر مثل اللغة المحكية. من الممكن كتابة المثال السابق بأسلوب حتى أفضل، مثل هذا:

مثال أفضل بلغة البرمجة فيجول بيسك عن اختبار لـ True وFalse بشكل ضمني.

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( Not printerError ) Then InitializePrinter()
If ( printerError ) Then NotifyUserOfError()
If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected ) Then PrintSummary()
If ( Not printerError ) Then CleanupPrinter()
```

إذا لم تدعم لغتك البرمجية المتغيرات البوليانية،[1] وعليك أن تحاكي هذه المتغيرات، فليس من الممكن أن تكون قادر على استخدام هذه التقنية، لأن محاكاة true وfalse، لا يستطيع دائمًا أن يتم اختباره مع العبارات البرمجية مثل

`while ( not done)`.

جعل التعابير المعقدة أبسط

يمكنك اتخاذ العديد من الخطوات لتبسيط التعابير المعقدة:

قسّم الاختبارات المعقدة إلى اختبارات جزئية مع متغيرات منطقية جديدة. بدلًا من إنشاء اختبار ضخم باستخدام نص دزينة من العناصر، اسند القيم الوسيطة إلى عناصر، تسمح لك بإنجاز اختبار أبسط.

انقل التعابير المعقدة إلى داخل توابع منطقية. إذا تم تكرار اختبار ما غالبًا، أو إذا كان هذا الاختبار يصرف الانتباه عن التدفق الأساسي للبرنامج، عندها انقل الشفرة من الاختبار إلى تابع، وقُم باختبار قيمة هذا التابع. على سبيل المثال، يوجد هنا اختبار مُعقد:

مثال بلغة البرمجة فيجول بيسك عن اختبار مُعقد.

```
If ( ( document.AtEndOfStream ) And ( Not inputError ) ) And _
```

[1] **إشارة مرجعية:** للمزيد من التفاصيل انظر القسم 12.5، "المتغيرات البوليانية"

```
 ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
 ( Not ErrorProcessing( ) ) Then
 ' do something or other
 ...
End If
```

هذا عبارة عن اختبار بشع عند قراءته إذا لم تكن مُهتم بالاختبار بحد ذاته. بوضع هذا الاختبار في تابع منطقي، يمكنك أن تعزل الاختبار وتسمح للقارئ نسيانه إلا إذا كان مُهم. فيما يلي مثال عن وضع اختبار if في تابع:

مثال بلغة البرمجة فيجول بيسيك عن اختبار مُعقد، تم نقله إلى تابع منطقي، باستخدام متغيرات وسيطة لجعل الاختبار أوضح.[1]

```
Function DocumentIsValid( _
 ByRef documentToCheck As Document ،_
 lineCount As Integer ،_
 inputError As Boolean _
 ) As Boolean
  Dim allDataRead As Boolean
 Dim legalLineCount As Boolean
 allDataRead = ( documentToCheck.AtEndOfStream ) And ( Not inputError )
 legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <=
MAX_LINES )
 DocumentIsValid = allDataRead And legalLineCount And ( Not
ErrorProcessing() )
End Function
```

تم تقديم متغيرات وسيطة هنا لتوضيح الاختبار على السطر الأخير تحت

يفترض هذا المثال أن () ErrorProcessingهو تابع منطقي، يُحدد حالة المعالجة الحالية. الآن، عندما تقوم بقراءة التدفق الأساسي للشفرة، فلن تحتاج لقراءة الاختبار المُعقّد:

مثال بلغة البرمجة فيجول بيسيك عن التدفق الأساسي للشفرة بدون الاختبار المُعقد.

```
If ( DocumentIsValid( document ،lineCount ،inputError ) ) Then
 ' do something or other
 ...
End If
```

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن تقنية استخدام المتغيرات الوسيطة لتوضيح اختبار منطقي، انظر "استخدم المتغيرات المنطقية لتوثيق برنامجك" في القيم 5.12



إذا كنت تستخدم الاختبار فقط مرة واحدة، فمن الممكن أن لا تفكر بأن وضع الاختبار في إجرائية بأمر جدير بالاهتمام. ولكن وضع الاختبار في تابع مُسمى بشكل جيد يُحسن من قابلية القراءة ويجعل من السهل رؤية آلية عمل شفرتك، وهذا سبب كاف للقيام بذلك.

يُقدم اسم التابع تجريدية داخل البرنامج، التي توثّق الغرض من الاختبار في الشفرة. إن هذا حتى أفضل من توثيق الاختبار باستخدام التعليقات، لأنه احتمال قراءة الشفرة أكبر من احتمال قراءة التعليقات، واحتمال تحديث الشفرة بشكل دائم أيضًا أكبر.

استخدم جداول القرار لاستبدال الشروط المُعقدة[1]. في بعض الأحيان، قد يكون لديك اختبار معقد يتضمن عدة متغيرات. قد يكون من المفيد استخدام جدول قرار لإنجاز الاختبار، بدلًا من استخدام عبارات `if` أو `case`. حيث قد يكون أسهل كتابة الشفرة بشكل بدائي للبحث في جدول قرار، فقط باستخدام بضعة أزواج من سطور الشفرة وبدون استخدام بنى التحكم. يُقلل هذا التصغير في التعقيد من فرصة ارتكاب أخطاء. وإذا كانت تتغير بياناتك، فيمكنك أن تغير جدول القرار بدون تغيير الشفرة؛ فقط تحتاج إلى تحديث محتوى هياكل البيانات.

صياغة التعابير المنطقية بشكل إيجابي

لدى معظم الناس مشاكل بفهم الكثير من (السلبية) في العبارة الواحدة. يمكنك القيام بالعديد من الأشياء لتجنب التعابير المنطقية السلبية المعقدة في برنامجك:

في العبارات البرمجية `if`، حوّل السلبيات إلى إيجابيات، واقلب الشفرة في حالات `if` و`else`. فيما يلي مثال عن اختبار مُعبّر عنه بشكل سلبي:

مثال بلغة البرمجة جافا عن اختبار منطقي سلبي مُربك.

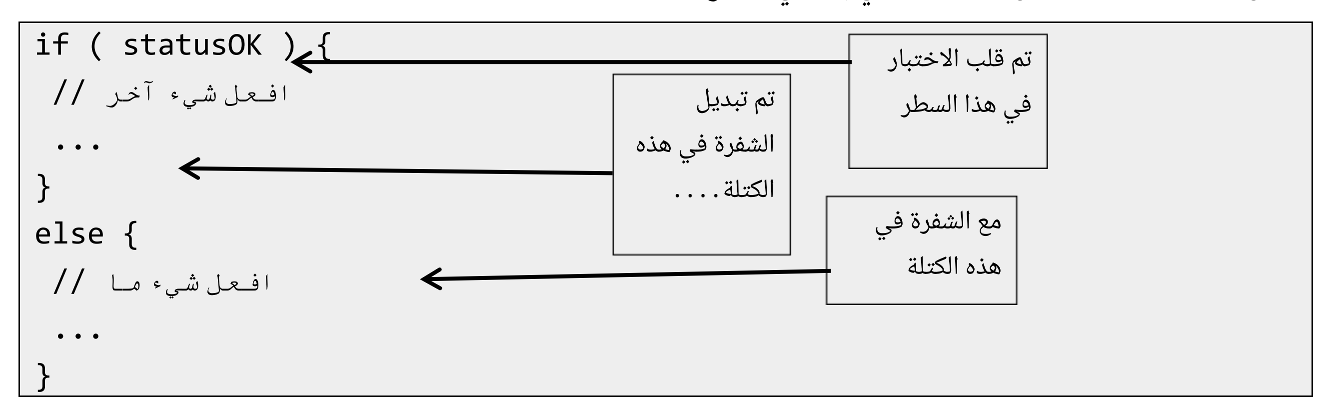
```
if ( !statusOK ) {
  // افعل شيء ما
  ...
}
else {
  // وإلا افعل شيء آخر
  ...
}
```

هنا العبارة السلبية (!) not

يمكنك استبدال هذا بالاختبار التالي المُعبر عنه بشكل إيجابي:

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول استخدام الجداول كبدائل عن المنطق المُعقد، انظر الفصل 18، "المناهج المُقادة بواسطة الجداول"

مثال بلغة البرمجة جافا عن اختبار منطقي إيجابي أوضح



إن مقطع الشفرة الثاني منطقيًا هو نفسه الأول ولكن من الأسهل قراءته[1]، لأنه تم استبدال التعابير السلبية بتعابير ايجابية.

بدلًا من ذلك، يمكنك اختيار اسم متغير مُختلف، يمكنه أن يعكس القيمة الحقيقية للاختبار. في المثال، يمكنك استبدال المتغير `statusOK` بالمتغير `ErrorDetected`، الذي يمكنه أن يكون صحيح عندما المتغير `statusOK` خاطئ.

طبّق نظريات DeMorgan لتبسيط الاختبارات المنطقية التي تستخدم التعابير السلبية. تسمح لك نظريات DeMorgan باستغلال العلاقة المنطقية بين تعبير ما ونسخته التي تعني الشيء نفسه، لأنه يتم ابطال التعبير مرتين. على سبيل المثال، من الممكن أن يكون لديك مقطع شفرة يحوي الاختبار التالي:

مثال بلغة البرمجة جافا لاختبار سلبي

```
if ( !displayOK || !printerOK ) ...
```

وفيما يلي المُكافئ المنطقي له:

مثال بلغة البرمجة جافا، بعد تطبيق نظريات DeMorgan

```
if ( !( displayOK && printerOK ) ) ...
```

هنا لا يوجد حاجة لقلب حالات `if` و`else`؛ إن التعابير في أخر مقطعين من الشفرة منطقيًا متكافئين. لتطبيق نظريات DeMorgan على المعامل المنطقية `and`، أو على المعامل المنطقية `or`، أو على كلا المعاملين، عليك نفي كل من عوامل المعامل المنطقي `and`، وتبديل `and` و`or`، ونفي كامل التعبير. يُلخص الجدول 19-1 عمليات التحويل الممكنة باستخدام نظريات DeMorgan.

---

[1] إشارة مرجعية: في بعض الأحيان تتعارض النصائح لصياغة تعابير منطقية بشكل إيجابي مع نصائح كتابة شفرة الحالة الإسمية بعد `if` بدلًا من بعد `else`- انظر القسم `1.15`، "العبارات البرمجية `if`"، في هكذا حالة، عليك أن تفكر بالفوائد لكل طريقة وتقرر أي من الطريقتين أفضل لحالتك.

الجدول 19-1 عمليات التحويل للتعابير المنطقية باستخدام نظريات DeMorgan.

التعبير الأولي	التعبير المكافئ
`not A and not B`	`not ( A or B )`
`not A and B`	`not ( A or not B )`
`A and not B`	`not ( not A or B )`
`A and B`	`not ( not A or not B )`
`not A or not B*`	`not ( A and B )`
`not A or B`	`not ( A and not B )`
`A or not B`	`not ( not A and B )`
`A or B`	`not ( not A and not B )`

* تم استخدام هذا التعبير في المثال

استخدام الأقواس لتوضيح التعابير المنطقية

إذا كان لديك تعبير منطقي مُعقد، فبدلًا من الاعتماد على ترتيب تقييم لغة البرمجة،[1] يمكنك استخدام الأقواس لجعل المعنى أوضح. يُقلل استخدام الأقواس من الاعتماد على القارئ، الذي من الممكن لا يفهم خفايا كيفية تقييم لغة البرمجة التي يستخدمها للتعابير المنطقية. إذا كنت ذكي، فلن تعتمد على نفسك أو على القارئ في الفهم العميق لأسبقية التقييم- وبشكل أساسي عندما عليك أن تبدل بين لغتين أو أكثر. لا يشبه استخدام الأقواس إرسال برقية: حيث لن يتم محاسبتك على كل حرف- حيث الأحرف الإضافية مجانية.

فيما يلي تعبير منطقي باستخدام القليل من الأقواس:

مثال بلغة البرمجة جافا عن تعبير يحوي القليل من الأقواس.

```
if ( a < b == c == d ) ...
```

إن هذا تعبير مُربك عند العمل معه، وهو حتى مُربك بشكل كبير لأنه من غير الواضح ماذا يعني كاتب الشفرة بالاختبار

`( a < b ) == ( c == d ) or ( ( a < b ) == c ) == d.`

لا تزال النسخة المُعدلة التالية من التعبير مُربكة بشكل قليل، ولكن هنا الأقواس تُساعد:

مثال بلغة البرمجة جافا عن تعبير أفضل باستخدام الأقواس.

```
if ( ( a < b ) == ( c == d ) ) ...
```

في هذه الحالة، تُساعد الأقواس على قابلية القراءة وعلى صحة البرنامج- حيث لن يقوم المترجم البرمجي بتفسير المقطع الأول من الشفرة بهذه الطريقة. عندما تقع بحالة شك، استخدم الأقواس.

---

1 **إشارة مرجعية**: للمزيد حول مثال استخدام الأقواس لتوضيح الأنواع الأخرى من التعابير، انظر "الأقواس"، في القسم 31.2.

استخدم تقنية عد بسيطة لموازنة الأقواس[1]. إذا كان لديك مشكلة بتحديد فيما إذا كان عدد الأقواس متوازن، فيما يلي نُقدم خدعة عد بسيطة تُساعد على هذا. ابدأ بقول "صفر". انتقل على طول التعبير، من اليسار إلى اليمين. وعندما تواجه قوس مفتوح قُل "واحد". في كل مرة تواجه قوس مفتوح آخر، قُم بزيادة العدد الذي تقوله. وفي كل مرة تواجه قوس مُغلق، أنقص من العدد الذي تقوله.وإذا، في نهاية التعبير، عُدتا إلى الصفر، فهذا يعني أن عدد أقواسك متوازن.

مثال بلغة البرمجة جافا عن الأقواس المتوازنة.

```
if ( ( ( a < b ) == ( c == d ) ) && !done ) ...   ← اقرأ هذا
 | | |   | |   | |     |
0 1 2 3   2 3   2 1     0        ←          قُل هذا
```

في هذا المثال، ينتهي العد مع العدد 0، ولهذا فإن عدد الأقواس متوازن. في المثال التالي، إن الأقواس غير متوازنة:

مثال بلغة البرمجة جافا عن الأقواس غير المتوازنة.

```
if ( ( a < b ) == ( c == d ) ) && !done ) ...
 | |   | |   | |    |          ←    اقرأ هذا
0 1 2   1   2   1 0    -1   ←    قُل هذا
```

إن العدد 0 الذي حصلت عليه قبل أخر قوس مُغلق هو عبارة عن نصيحة بأنه يوجد قوس مفقود قبل هذه النقطة. حيث من غير المفروض أن تحصل على 0 حتى أخر قوس من التعبير.

ضع الأقواس بشكل كامل للعبارات المنطقية. إن الأقواس رخيصة، وهم عامل مساعد على زيادة قابلية القراءة. إن عادة وضع الأقواس بشكل كامل في العبارات المنطقية هي عبارة عن مماسة عملية جيدة.

معرفة كيف يتم تقييم التعابير المنطقية

لدى العديد من لغات البرمجة صيغة ضمنية للتحكم، تلعب دور في تقييم التعابير المنطقية. تقوم بعض المترجمات البرمجية في بعض اللغات البرمجية بتقييم كل عنصر من التعبير المنطقي قبل ترجمة العناصر وتقييم كامل التعبير. ولدى مترجمات اللغات الأخرى "دورة قصيرة" أو "كسولة" من التقييم، حيث أنها تُقيم فقط القطع الضرورية. إن هذا مُهم بشكل خاص، حيث بالاعتماد على نتائج الاختبار الأول، من الممكن ألا ترغب في إجراء الاختبار الثاني. على سبيل المثال، افترض أنك تتفحص عناصر مصفوفة ولديك الاختبار التالي:

---

[1] **إشارة مرجعية:** تملك العديد من المحررات البرمجية أوامر تُقابل جميع أنواع الأقواس. لمزيد من التفاصيل حول المحررات البرمجية، انظر "التحرير" في القسم 2.30.

مثال شفرة مزيفة عن اختبار خاطئ.

```
while ( i < MAX_ELEMENTS and item[ i ] <> 0 ) ...
```

إذا تم تقييم كامل التعبير، فإنك ستحصل على خطأ في المرور الأخير للحلقة. يساوي المتغير i قيمة المتغير maxElements، فلهذا التعبير item[i] مُكافئ للتعبير item [maxElements]، الذي هو عبارة عن خطأ في فهرس المصفوفة. من الممكن أن تجادل بأن هذا ليس بمشكلة، طالما أنك فقط تبحث عن القيمة، وليس عن تغييرها. ولكن هذا ممارسة عملية غامضة ومن الممكن أن تربك الشخص القارئ للشفرة. في كثير من البيئات، سيولد هذا دائما إما خطأ في وقت التشغيل أو انتهاك للحماية.

في الشفرة الزائفة، يمكنك أن تُعيد بناء الاختبار بطريقة لا يحدث فيها خطأ:

مثال شفرة مزيفة عن اختبار مُعاد بناءه بشكل صحيح

```
while ( i < MAX_ELEMENTS )
 if ( item[ i ] <> 0 ) then
 ...
```

إن هذا صحيح لأنه لا يتم تقييم التعبير item[i]، إلا إذا كان i أصغر من maxElements.

تؤمن العديد من لغات البرمجة الحديثة التسهيلات التي تمنع حدوث هذا النوع من الخطأ في المكان الأول. على سبيل المثال، تستخدم لغة البرمجة سي++ دورة التقييم القصيرة: إذا كان المعامل الأول من and خاطئ(False)، فلن يتم تقييم المعامل الثاني، لأنه عندها سيكون كامل التعبير خاطئ على أية حال. بكلمات أخرى، في سي++ الجزء الوحيد من:

```
if ( SomethingFalse && SomeCondition ) ...
```

الذي يتم تقيمه هو SomethingFalse. حيث تتوقف عملية التقييم بمجرد أن SomethingFalse مُعرف كقيمة خاطئة. إن تقييم العملية or أيضًا بشكل مشابه هو عبارة عن تقييم الدورة القصيرة. في سي++ وجافا، الجزء الوحيد من:

```
if ( somethingTrue || someCondition ) ...
```

الذي سيتم تقيمه هو somethingTrue. حيث تتوقف عملية التقييم بمجرد أن somethingTrue مُعرفة كقيمة صحيحة (True)، لأن التعبير المنطقي هذا دائما صحيح إذا كان أي جزء منه صحيح. كنتيجة لطريقة التقييم هذه، إن العبارات البرمجية التالية هي عبارات مقبولة.

مثال بلغة البرمجة جافا عن اختبار يعمل لأن دورة التقييم قصيرة

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) )
...
```

إذا تم تقييم التعبير بشكل كامل عندما denominator تساوي0، ستنتج العملية في السطر الثاني خطأ التقسيم على صفر. ولكن بما أنه لا يتم تقييم الجزي الثاني إلا إذا كان الجزء الأول صحيح، فإنه لن يتم أبداً تقيمه عندما denominator تساوي 0، وبالتالي لن يحدث خطأ التقسيم على صفر.

من ناحية أخرى، بما أنه يتم تقييم العملية && (and) من اليسار إلى اليمين، فإنه بشكل منطقي لن تعمل عبارة التقييم التالية:

مثال بلغة البرمجة جافا عن اختبار لا ينقذ دورة التقييم القصيرة.

```
if ( ( ( item / denominator ) > MIN_VALUE ) && ( denominator != 0 ) )
...
```

في هذه الحالة، يتم تقييم item / denominator قبل denominator != 0. وبالنتيجة تقترف هذه الشفرة خطأ التقسيم على صفر.

تُعقّد لغة البرمجة جافا هذا الوضع بتوفيرها للعوامل "المنطقية ". تضمن العوامل المنطقية في جافا & و| بأنه سيتم تقييم جميع العناصر في التعابير المنطقية بغض النظر عن إمكانية تحديد صحة أو خطأ التعبير بدون التقييم الكامل له. بكلمات أخرى، في لغة البرمجة جافا، هذا أمن:

مثال بلغة البرمجة جافا عن اختبار يعمل لأن دورة التقييم (الشرطية) قصيرة.

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) )
...
```

ولكن هذا غير أمن:

مثال بلغة البرمجة جافا عن اختبار لا يعمل لأن الدورة القصيرة للتقييم غير مكفولة.

```
if ( ( denominator != 0 ) & ( ( item / denominator ) > MIN_VALUE ) )
...
```



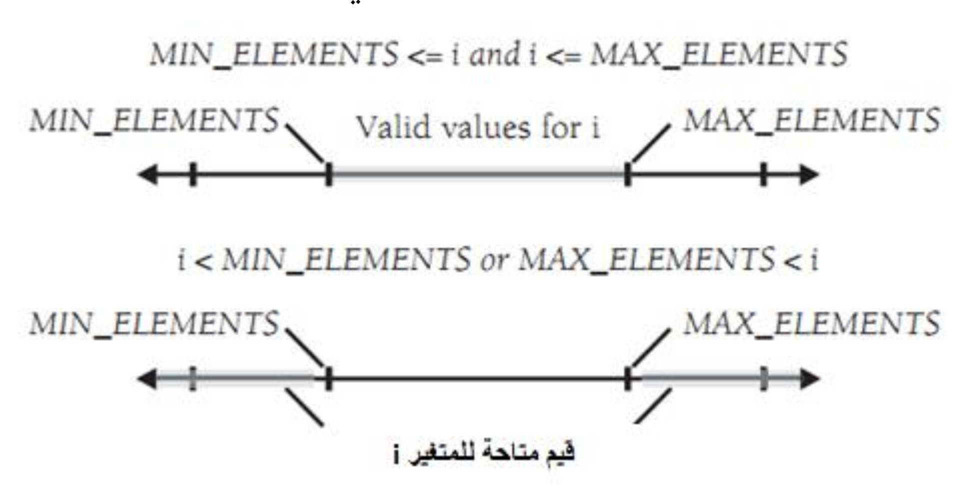
تستخدم لغات برمجة مختلفة أنواع مختلفة من التقييم، وتميل منفذات اللغات إلى تقديم الحرية في تقييم التعبير، لذلك تفحّص الدليل لنسخة مخصصة من لغة البرمجة التي تستخدمها لإيجاد أي نوع من التقييم تستخدمه لغة البرمجة. الأفضل حتى الآن، طالما يوجد إمكانية لعدم كون القارئ لشفرتك حاد التفكير مثلك، استخدام الاختبارات المتداخلة لتوضيح نواياك بدلًا من الاعتماد على ترتيب التقييم والتقييم ذو الدورة القصيرة.

كتابة التعبيرات الرقمية في ترتيب رقم السطر

نظّم الاختبارات الرقمية بحيث تلي النقاط على رقم السطر. بشكل عام، ابني اختباراتك الرقمية بحيث يكون لديك مقارنات كالتالي:

```
MIN_ELEMENTS <= i and i <= MAX_ELEMENTS
i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

الفكرة بترتيب العناصر من اليسار إلى اليمين، من الأصغر إلى الأكبر. في السطر الأول، إن MIN_ELEMENTS وMAX_ELEMENTS كلاهما نقاط نهاية، لهذا أنهم يستمرون عند النهاية. من المفروض أن يكون المتغير i بين هذين العنصرين، لذلك هو يستمر عند المنتصف. في المثال الثاني، أنت تختبر فيما إذا كان المتغير i خارج المجال، ولهذا فإنه يستمر خارج الاختبار عند كلا النهايتين، ويستمر كلا MIN_ELEMENTS وMAX_ELEMENTS في الداخل. ثقابل هذه الطريقة بسهولة الصورة المرئية للمقارنة في الشكل 19-1:



الشكل 19-1 أمثلة عن استخدام ترتيب رقم السطر للاختبارات المنطقية.

إذا كنت تختبر المتغير i فقط مُقابل MIN_ELEMENTS، فإنه يتغير موقع i بالاعتماد أين كان i عندما الاختبار كان ناجح. إذا كان من المفروض أن تكون i أصغر، فسيوف يكون لديك اختبار كالتالي:

```
while ( i < MIN_ELEMENTS ) ...
```

ولكن إذا كان من المفترض أن يكون المتغير i أكبر، فسيوف يكون لديك اختبار كالتالي:

```
while ( MIN_ELEMENTS < i ) ...
```

هذه الطريقة أوضح من الاختبارات التالية:

```
( i > MIN_ELEMENTS ) and ( i < MAX_ELEMENTS )
```

التي لا تعطي القارئ أية مساعدة في تصور ماذا يتم اختباره.

إرشادات توجيهية للمقارنات مع

تستخدم لغات البرمجة العدد 0 لأغراض متعددة. إنه قيمة عددية. إنه فاصل فراغ في سلسلة محرفية. إنه قيمة للمؤشر الذي لا يؤشر إلى أي شيء (null). إنه قيمة للعنصر الأول في تعداد. إنه خطأ false في التعابير

المنطقية. بما أن 0 يُستخدم للعديد من الأغراض، لذلك عليك أن تكتب شفرة، تُسلط الضوء على الطريقة الخاصة التي يتم فيها استخدام 0.

قارن المتغيرات المنطقية بشكل ضمني. كما تم ذكر هذا سابقًا، من المُناسب كتابة التعابير المنطقية كما يلي:

```
while ( !done ) ...
```

إن هذه المقارنة الضمنية لـ 0 مُناسبة لأن المقارنة هي داخل تعبير منطقي.

قارن الأعداد مع العدد 0. على الرغم من أنه من المناسب مقارنة التعابير المنطقية بشكل ضمني، عليك أن تقارن التعابير العددية بشكل صريح (غير ضمني). على سبيل المثال، اكتب:

```
while ( balance != 0 ) ...
```

بدلًا من كتابة

```
while ( balance ) ...
```

قارن المحارف مع فاصل الفراغ `('\0')` ***بشكل صريح في لغة البرمجة سي.*** إن المحارف مثل الأرقام ليست بتعابير منطقية. ولهذا، من أجل المحارف، اكتب:

```
while ( *charPtr != '\0' ) ...
```

بدلًا من كتابة

```
while ( *charPtr ) ...
```

تُعارض هذه النصيحة الاتفاقية الشائعة في لغة البرمجة سي للتعامل مع البيانات المحرفية (كما في المثال الثاني هنا)، ولكن تُعزز هذه النصيحة فكرة أن التعبير يعمل مع بيانات محرفية بدلًا من بيانات منطقية. لا تعتمد بعض اتفاقيات لغة البرمجة سي على زيادة قابلية القراءة أو زيادة قابلية الصيانة، وهذا مثال عنها. لحسن الحظ، تتلاشى القضية كلها عند كتابة الشفرة باستخدام سي++ ومحارف STL.

قارن المؤشرات مع NULL. من أجل المؤشرات اكتب

```
while ( bufferPtr != NULL ) ...
```

بدلًا من كتابة

```
while ( bufferPtr ) ...
```

كما في التوصية من أجل المحارف، تُعارض هذه التوصية اتفاقية لغة البرمجة سي الأساسية، ولكن يُبرر هذا الربح في قابلية القراءة.

المشاكل الشائعة في التعابير المنطقية

إن التعابير المنطقية هي عبارة عن سبب لمشاكل إضافية متعلقة بلغة برمجة محددة:

في لغات البرمجة المستمدة من لغة البرمجة سي، ضع الثوابت على الجانب الأيسر من المقارنات. تُعاني لغات البرمجة المستمدة من لغة البرمجة سي من مشاكل خاصة في التعابير المنطقية. إذا كانت لديك مشاكل في كتابة = بدلًا من ==، خذ بعين الاعتبار الاتفاقية البرمجية لوضع الثوابت والمحارف على الجزء الأيسر من التعابير المنطقية، كما يلي:

مثال بلغة البرمجة سي ++ لوضع الثابت على الجزء الأيسر من التعبير المنطقي- حيث سيلتقط المجمع خطأ.

```
if ( MIN_ELEMENTS = i ) ...
```

في هذا التعبير، يجب على المترجم البرمجي أن يُحدد الإشارة = كخطأ، طالما أن إسناد أي شيء إلى ثابت هو خطأ. بشكل مُعاكس، في التعبير التالي، سيُحدد المترجم البرمجي هذا فقط كتحذير، وفقط إذا كانت تحذيرات المترجم البرمجي مُفعلّة:

مثال بلغة البرمجة سي ++ لوضع الثابت على الجزء الأيمن من التعبير المنطقي- حيث من الممكن ألا يلتقط المترجم البرمجي أية أخطاء.

```
if ( i = MIN_ELEMENTS ) ...
```

تتعارض هذه النصيحة مع نصيحة استخدام ترتيب رقم السطر. إن خياري الشخصي المُفضل هو استخدام ترتيب رقم السطر والسماح للمترجم البرمجي بتحذيري حول عمليات الإسناد غير المقصودة.

في لغة البرمجة سي++، خُذ بعين الاعتبار إنشاء بدائل الماكرو من أجل &&، ||، و== (ولكن فقط كحل أخير). إذا كان لديك مثل هكذا مُشكلة، من الممكن إنشاء تعريف #define للماكرو and وor، ومن المكن استخدام AND وOR بدلًا من && و||. وبشكل مُشابه إن استخدام = في المكان الذي تعني فيه استخدام == هو خطأ سهل ارتكابه. إذا ارتكبت هذا الخطأ عدة مرات، فمن الممكن أن تُنشأ ماكرو مثل EQUALS لعملية مقارنة التساوي (==).

يرى العديد من المبرمجين الخُبراء هذه الطريقة مُساعد على قابلية القراءة للمُبرمج، الذي لا يمكنه أن يُحافظ على تفاصيل لغة البرمجة بشكل مُباشر، وكقابلية للقراءة مُهينة للمبرمج الاكثر طلاقة في التعامل مع لغة البرمجة. إضافة إلى أنه تزود مُعظم المترجمات البرمجية تحذيرات الخطأ لاستخدام عملية الإسناد والعمليات الثنائية التي تبدو كأخطاء. إن تفعيل تحذيرات المترجم البرمجي هو عادةً الخيار الأفضل من إنشاء ماكرو غير قياسي.

في لغة البرمجة جافا، اعرف الفرق بين a==b وa.equals(b). في لغة البرمجة جافا، إن اختبارات a==b هي من أجل معرفة فيما إذا كان a وb يُشيران إلى نفس الكائن، بينما اختبارات a.equals(b) هي من أجل

معرفة فيما إذا كانت الكائنات تملُك نفس القيمة المنطقية. بشكل عام، يجب على برامج جافا أن تستخدم تعابير مثل `a.equals(b)` بدلًا من `a==b`.

2.19 العبارات البرمجية المركبة (الكتل/البلوكات)

إن "العبارة المُعقدة" أو "البلوك" هو عبارة عن تجميع مجموعة من العبارات التي يتم مُعالجتها كعبارة وحيدة من أجل أغراض التحكم بالتدفق لبرنامج. يتم انشاء العبارات المُعقدة باستخدام الأقواس { و} حول مجموعة العبارات في سي++ وC# وسي وجافا. في بعض الأحيان يتم تضمينها في كلمات مفتاحية لأمر، مثل `For` أو `Next` في لغة البرمجة فيجول بيسك. فيما يلي الإرشادات التوجيهية لاستخدام العبارات المُعقدة بشكل فعّال:

اكتب أزواج الأقواس مع بعضهم البعض[1]. املأ الوسط بين القوسين بعد أن تكون قد كتبتهما كلاهما (قوس الفتح وقوس الإغلاق). يشتكي بعض الناس من صعوبة مُقابلة الأقواس أو بداية زوج الأقواس ونهايته، وهذه تمامًا ليست بمشكلة. إذ كنت تتبع هذه النصيحة، فلن يكون لديك أبدًا مشكلة في مقابلة الأقواس.

اكتب هذا أولًا:

```
for ( i = 0; i < maxLines; i++ )
```

ثم اكتب هذا:

```
for ( i = 0; i < maxLines; i++ ) { }
```

وأخيرًا اكتب هذا:

```
for ( i = 0; i < maxLines; i++ ) {
 // whatever goes in here ...
}
```

يتم تطبيق هذه الطريقة على كل هياكل البلوكات، بما فيها `if` و`for` و`while` في سي++ وجافا، و`If-Then-Else` و`For-Next` ومقارنات `While-Wend` في فيجول بيسك.

استخدم الأقواس لتوضيح الشروط. من الصعوبة جدًا قراءة الشروط بدون تحديد أية عبارات هي مع اختبار `if`. إن وضع عبارة وحيدة بعد اختبار `if` هو في بعض الأحيان جذاب من الناحية الجمالية، ولكن عند الصيانة، تميل مثل هذه العبارات إلى أن تصبح كتلًا أكثر تعقيدً، وإن العبارات المُفردة هي أكثر عرضة للخطأ عندما يحدث هذا.

[1] **إشارة مرجعية**: تملك العديد من المحررات البرمجية أوامر تقوم بمقابلة الاقواس. لمزيد من التفاصيل، انظر "التحرير"، في القسم 2.30.

استخدم البلوكات لتوضيح نواياك بغض النظر عما إذا كانت الشفرة داخل الكتلة هي سطر وحيد أو عشرين سطر.

3.19 العبارات البرمجية الفارغة

في لغة البرمجة سي++، من الممكن أن يوجد عبارات فارغة، عبارة تتكون بشكل كامل من فاصلة منقوطة، كما فيما يلي:

مثال بلغة البرمجة سي++ عن عبارة فارغة تقليدية

```
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() )
 ;
```

تتطلب الحلقة `while` في سي++ أن يتم إتباعها بعبارة برمجية، ولكن من الممكن أن تكون هذه العبارة هي عبارة فارغة. تمثل الفاصلة المنقوطة لوحدها على سطر العبارة الفارغة. فيما يلي إرشادات توجيهية للتعامل مع العبارات البرمجية الفارغة في س++:

لفت الانتباه إلى العبارات الفارغة[1]. إن العبارات البرمجية الفارغة غير شائعة، لذلك اجعلهم واضحين. إحدى طرق توضيحها هي بإعطاء الفاصلة المنقوطة للعبارة الفارغة سطر كامل خاص بها. ضع مسافة بادئة لها، تمامًا كما تفعل مع أي عبارة برمجية أخرى. إن هذه الطريقة موضحة في المثال السابق. كحل بديل، يمكنك أن تستخدم مجموعة من الأقواس الفارغة للتأكيد على العبارة الفارغة. فيما يلي مثالين:

مثال بلغة البرمجة سي++ عن عبارة فارغة يتم التأكيد عليها.

```
while ( recordArray.Read( index++ ) ) != recordArray.EmptyRecord() ) {
}
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {
 ;
}
```

هذه طريقة أولى لإظهار العبارة الفارغة.

هذه هي طريقة ثانية لإظهار العبارة الفارغة.

أنشأ الماكرو المُعالج `DoNothing()` أو تابع ضمني للعبارات الفارغة. لا تقوم العبارة الفارغة بأي شيء، ولكنها تجعل حقيقة افتراض عدم القيام بأي شيء واضحة بشكل مؤكد. هذا مُشابه لوضع علامة على الصفحات

[1] **إشارة مرجعية**: إن الطريقة الأفضل للتعامل مع العبارات الفارغة هي على الأرجح بتجنبهم. لمزيد من التفاصيل، انظر "تجنب الحلقات الفارغة" في القسم 2.16.

الفارغة باستخدام العبارة " هذه الصفحة تُركت فارغة عمدًا". إن الصفحة ليست بالحقيقة فارغة، ولكنك تعرف أنه من المفروض أنه لا يوجد أي شيء أخر عليها.

فيما يلي مشروح كيفية صناعة العبارات الفارغة في سي++ باستخدام #define. يمكنك أيضًا إنشاءها باستخدام التوابع الضمنية، التي سيكون لديها نفس التأثير.

مثال بلغة البرمجة سي++ عن عبارة فارغة يتم التأكيد عليها باستخدام المالكرو ()DoNothing.

```
#define DoNothing()
...
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {
 DoNothing();
}
```

بالإضافة لاستخدام ()DoNothing في حلقات while و for الفارغة، تستطيع استخدامه للخيارات غير المهمة في عبارة switch. يوضح تضمين ()DoNothing أن هذه الحالة تم أخذها ولكن من المفروض أن لا تقوم بأي شيء.

إذا لم تكن تدعم لغة البرمجة الماكرو أو التوابع الضمنية، يمكنك إنشاء إجرائية ()DoNothing، التي بشكل مباشر تُعيد التحكم إلى الإجرائية التي قامت باستدعائها.

فكّر فيما إذا ستكون الشفرة أوضح باستخدام جسم حلقة غير فارغ. إن معظم الشفرة التي تُنتج حلقات بأجسام فارغة، تعتمد على الأثار الجانبية في شفرة التحكم بالحلقة. في معظم الحالات، إن الشفرة أكثر قابلية على القراءة، عندما تكون الآثار الجانبية واضحة، كما هو موضح:

أمثلة بلغة البرمجة سي++ عن شفرة مُعاد كتابتها بشكل أوضح باستخدام جسم حلقة غير فارغ.

```
RecordType record = recordArray.Read( index );
index++;
while ( record != recordArray.EmptyRecord() ) {
 record = recordArray.Read( index );
 index++;
}
```

تُقدم هذه الطريقة متغير إضافي للتحكم بالحلقة، وتتطلب سطور إضافية من الشفرة، ولكنها تُؤكد على الممارسة البرمجية البسيطة بدلًا من ذكاء استخدام الأثار الجانبية. مثل هكذا تأكيد مُفضل في شفرة الإنتاج.

4.19 ترويض التداخل العميق الخطير

لقد تم شهر "التداخل" في الأدب الحاسوبي لمدة 25 سنة، وما يزال المتهم الأساسي في الشفرة المُربكة. اقترحت دراسات نعوم تشومسكي وجيرالد واينبرغ أن قلة من الناس يمكنهم فهم أكثر من ثلاثة مستويات متداخلة من عبارات `if` (يوردون 1986)، ويوصي العديد من الباحثين بتجنب التداخل لأكثر من ثلاثة أو أربعة مستويات (مايرز 1976، ماركا 1981، ليدجارد وتاوير 1987). يعمل التداخل العميق ضد ما يصفه الفصل 5 "التصميم في عملية البناء" كضرورة تقنية أولية للبرنامج: إدارة التعقيد. وهذا عبارة عن سبب كافي لتجنب التداخل العميق.



ليس من الصعب تجنب التداخل العميق. إذا كان لديك تداخل عميق، يمكنك إعادة تصميم الاختبارات في حالات `if` و`else`، أو يمكنك تفكيك الشفرة إلى إجرائيات أبسط. تُمثل الأقسام التالية طُرق متعددة لإنقاص عمق التداخل:

بسّط `if` المُتداخلة بإعادة اختبار جزء من الشرط. إذا كان التداخل عميق بشكل كبير، يمكنك إنقاص عدد مستويات التداخل بإعادة اختبار بضع من الشروط. لدى مثال الشفرة التالية تداخل عميق إلى درجة يصبح فيها إعادة الهيكلة أمر ضروري:





مثال بغة البرمجة سي++ عن شفرة مُتداخلة بشكل عميق سيء.[1]

```
if ( inputStatus == InputStatus_Success ) {
 // lots of code
 ...
 if ( printerRoutine != NULL ) {
 // lots of code
 ...
 if ( SetupPage() ) {
 // lots of code
 ...
 if ( AllocMem( &printData ) ) {
 // lots of code
 ...
 }
 }
```

[1] **إشارة مرجعية**: إن الجزء المُعاد اختباره من الشرط لتقليل التعقيد مُشابه لإعادة اختبار حالة متغير. يتم توضيح هذه التقنية في "معالجة الخطأ و`goto`" في القسم 3.17.

```
 }
}
```

تم اختلاق هذا المثال لتوضيح مستويات التداخل. المقصود من //lots of code هو الإشارة إلى أن الإجرائية تحتوي على شفرة كبيرة بشكل كافي لتمتد عبر عدة شاشات أو عبر حدود الصفحة المطبوعة. فيما يلي الشفرة المُنقّحة لاستخدام إعادة الاختبار بدلًا من التداخل:

مثال بلغة البرمجة سي++ لشفرة Mercifully غير المُتداخلة باستخدام إعادة الاختبار.

```
if ( inputStatus == InputStatus_Success ) {
 // lots of code
 ...
 if ( printerRoutine != NULL ) {
 // lots of code
 ...
 }
}
if ( ( inputStatus == InputStatus_Success ) &&
 ( printerRoutine != NULL ) && SetupPage() ) {
 // lots of code
 ...
 if ( AllocMem( &printData ) ) {
 // lots of code
 ...
 }
}
```

هذا عبارة عن مثال واقعي بشكل خاص، لأنه يُظهر بأنك لا تستطيع إنقاص مستوى التداخل بشكل مجاني؛ يجب عليك أن تضع اختبار أكثر تعقيداً من أجل إنقاص مستوى التداخل. إن التخفيض من أربع مستويات إلى مستويين هو تحسين كبير في قابلية القراءة، ومع ذلك هو جدير بالدراسة.

بسّط if المُتداخلة باستخدام تقنية تكسير الكتلة "البلوك" . بديل للطريقة التي تم وصفها للتو هو بتعريف قسم من الشفرة التي سيتم تنفيذها ككتلة. إذا فشلت بعض الشروط في منتصف البلوك، فإن التنفيذ سيتجاوزها إلى نهاية الكتلة.

مثال بلغة البرمجة سي++ لاستخدام تقنية كسر الكتلة.

```
do {
 // begin break block
 if ( inputStatus != InputStatus_Success ) {
```

```
break; // break out of block
}
// lots of code
...
if ( printerRoutine == NULL ) {
break; // break out of block
}
// lots of code
...
if ( !SetupPage() ) {
break; // break out of block
}
// lots of code
...
if ( !AllocMem( &printData ) ) {
break; // break out of block
}
// lots of code
...
} while (FALSE); // end break block
```

إن هذه الطريقة غير شائعة بما فيه الكفاية، حيث يجب ان يتم استخدامها فقط عندما كامل فريقك مُتألف معها وعندما يتم اعتمادها من قبل الفريق كممارسة كتابة شفرة مقبولة.

حوّل if المُتداخلة إلى مجموعة من if-then-elses. إذا كنت تفكر باختبار if المُتداخلة بشكل حرج، فإنه من الممكن أن تكتشف بأنك تستطيع أن تُدرك إمكانية استخدام if-then-else بدلًا من if المُتداخلة. افترض أنه لديك شجرة قرار كثيفة كالتالية:

مثال بلغة البرمجة جافا لشجرة قرار كثيفة.

```
if ( 10 < quantity ) {
if ( 100 < quantity ) {
if ( 1000 < quantity ) {
discount = 0.10;
}
else {
discount = 0.05;
```

```
  }
  }
  else {
  discount = 0.025;
  }
}
else {
  discount = 0.0;
}
```

إن هذا الاختبار مُنظم بشكل ضعيف بعدة طُرق، واحدة منها أنه يوجد اختبارات فائضة. عندما تختبر أية كمية أكبر من 1000، فإنه لن تكون حاجة لاختبار فيما إذا كانت القيمة أكبر من 100 وأكبر من 10. بالنتيجة، يمكنك إعادة تنظيم الشفرة كما يلي:

مثال بلغة البرمجة جافا لـ if مُتداخلة محولة إلى مجموعة من if-then-else.

```
if ( 1000 < quantity ) {
  discount = 0.10;
}
else if ( 100 < quantity ) {
  discount = 0.05;
}
else if ( 10 < quantity ) {
  discount = 0.025;
}
else {
  discount = 0;
}
```

هذا الحل أسهل من غيره، لأن الأعداد تزداد بشكل مُنظم. فيما يلي مُبين كيفية إعادة صياغة `if` المُتداخلة إذا لم تكن الأرقام مُنظمة:

مثال بلغة البرمجة جافا عن if مُتداخلة محولة إل مجموعة من if-then-else عندما الأرقام "فوضوية".

```
if ( 1000 < quantity ) {
  discount = 0.10;
}
else if ( ( 100 < quantity ) && ( quantity <= 1000 ) ) {
  discount = 0.05;
```

```
}
else if ( ( 10 < quantity ) && ( quantity <= 100 ) ) {
 discount = 0.025;
}
else if ( quantity <= 10 ) {
 discount = 0;
}
```

إن الفرق الأساسي بين هذه الشفرة والشفرة السابقة بأن التعابير في حالات `else-if` لا تعتمد على الاختبارات السابقة. لا تحتاج هذه الشفرة إلى حالات `else` لكي تعمل، ومن الممكن تنفيذ الاختبارات في أية ترتيب. من الممكن أن تحتوي الشفرة على أربع `if` بدون أية `else`. إن السبب الوحيد لتفضيل نسخة `else`، إنها تتجنب التكرار غير الضروري للاختبارات.

حوّل `if` المُتداخلة إلى عبارة حالة. يمكنك إعادة كتابة الشفرة لبعض أنواع الاختبارات، بشكل خاص تلك التي مع أعداد صحيحة، وذلك باستخدام عبارة الحالة `case` بدلًا من استخدام سلاسل `if` و`else`. لا يمكنك استخدام هذه التقنية في بعض لغات البرمجة، ولكنها تقنية فعّالة للغات التي تدعمها. فيما يلي موضح كيفية إعادة كتابة الشفرة في مثال فيجول بيسك:

مثال بلغة البرمجة فيجول بيسيك عن if مُتداخلة محولة إلى عبارة حالة.

```
Select Case quantity
 Case 0 To 10
 discount = 0.0
 Case 11 To 100
 discount = 0.025
 Case 101 To 1000
 discount = 0.05
 Case Else
 discount = 0.10
End Select
```

يُقرأ هذا المثال ككتاب. حيث عندما تقوم بمقارنة هذا المثال مع المثالين السابقين قبل عدة صفحات سابقًا، يبدو بشكل خاص حل واضح.

ضع الشفرة المتداخلة داخل إجرائية خاصة بها. إذا حدث تداخل عميق داخل حلقة، فإنك تستطيع غالبًا تحسين هذه الحالة بوضع الشفرة الموجودة داخل الحلقة في إجرائية خاصة بها. إن هذا فعّال خاصة عندما

التداخل هو نتيجة لاستخدام الشروط والحلقات. اترك فروع if-then-else في الحلقة الرئيسية لإظهار تفرّع القرار، ومن ثم انقل العبارات البرمجية داخل الفروع إلى الإجرائيات الخاصة بها. تحتاج هذه الشفرة إلى تحسين باستخدام تعديل كالتالي:

مثال لغة البرمجة سي++ لشفرة مُتداخلة، تحتاج إلى فصلها في إجرائيات.

```
while ( !TransactionsComplete() ) {
 // transaction قراءة سجل مناقلة
 transaction = ReadTransaction();
 // معالجة المناقلة، اعتمادا على نوعها
 if ( transaction.Type == TransactionType_Deposit ) {
 // Deposit معالجة الإيداع
 if ( transaction.AccountType == AccountType_Checking ) {
 if ( transaction.AccountSubType == AccountSubType_Business )
 MakeBusinessCheckDep( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountSubType == AccountSubType_Personal )
 MakePersonalCheckDep( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountSubType == AccountSubType_School )
 MakeSchoolCheckDep( transaction.AccountNum ،transaction.Amount );
 }
 else if ( transaction.AccountType == AccountType_Savings )
 MakeSavingsDep( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountType == AccountType_DebitCard )
 MakeDebitCardDep( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountType == AccountType_MoneyMarket )
 MakeMoneyMarketDep( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountType == AccountType_Cd )
 MakeCDDep( transaction.AccountNum ،transaction.Amount );
 }
 else if ( transaction.Type == TransactionType_Withdrawal ) {
 // معالجة السحب
 if ( transaction.AccountType == AccountType_Checking )
 MakeCheckingWithdrawal( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountType == AccountType_Savings )
 MakeSavingsWithdrawal( transaction.AccountNum ،transaction.Amount );
 else if ( transaction.AccountType == AccountType_DebitCard )
 MakeDebitCardWithdrawal( transaction.AccountNum ،transaction.Amount );
 }
```

```
else if ( transaction.Type == TransactionType_Transfer ) {
 MakeFundsTransfer(
 transaction.SourceAccountType ،
 transaction.TargetAccountType ،
 transaction.AccountNum ،
 transaction.Amount
 );
 }
 else {
 // معالجة الأنواع غير المعروفة من المناقلة
 LogTransactionError( "Unknown Transaction Type" ،transaction );
 }
}
```

هنا نوع الإجراء TransactionType_Transfer

على الرغم من أن الشفرة معقدة، فإنها ليست أسوء شفرة من الممكن أن تراها. حيث إنها متداخلة على أربع مستويات، وموثقة بتعليقات، وتملك غاية منطقية، والتحليل الوظيفي فيها كافي، خاصة بالنسبة لنوع الإجراء Transact ionType_Transfer. على الرغم من كفايتها، تستطيع أن تحسنها بفصل المحتوى لاختبارات if الداخلية في إجرائيات خاصة بها.

مثال لغة البرمجة سي++ لشفرة مُتداخلة جيدة بعد التحليل الوظيفي في إجرائيات.[1]

```
while ( !TransactionsComplete() ) {
 // transaction قراءة سجل
 transaction = ReadTransaction();
 // معالجة المناقلة، اعتمادا على نوعها
 if ( transaction.Type == TransactionType_Deposit ) {
 ProcessDeposit(
 transaction.AccountType ،
 transaction.AccountSubType،
 transaction.AccountNum ،
 transaction.Amount
 );
 }
 else if ( transaction.Type == TransactionType_Withdrawal ) {
 ProcessWithdrawal(
 transaction.AccountType ،
 transaction.AccountNum،
```

[1] **إشارة مرجعية:** إن هذا النوع من التحليل الوظيفي سهل بشكل خاص إذا قمت من البداية ببناء الإجرائية وفق الخطوات الموصوفة في الفصل 9 "عملية برمجة الشفرة المزيفة"، إن الإرشادات التوضيحية للتحليل الوظيفي مُقدمة في "فرق تَسُد" في القسم 5.4.

```
 transaction.Amount
 );
 }
 else if ( transaction.Type == TransactionType_Transfer ) {
 MakeFundsTransfer(
 transaction.SourceAccountType ،
 transaction.TargetAccountType،
 transaction.AccountNum ،
 transaction.Amount
 );
 }
 else {
 // غير المعروفة transaction معالجة أنواع
 LogTransactionError("Unknown Transaction Type" ،transaction );
 }
}
```

تمت إزالة الشفرة ببساطة من الإجرائية الأصلية ووضعها وصياغتها في الإجرائيات الجديدة. (لم يتم عرض الإجرائيات الجديدة هنا). لدى الشفرة الجديدة العديد من الحسنات. أولاً، يجعل تداخل مستوين الهيكلة أبسط وأسهل للفهم. ثانيًا، يمكنك أن تقرأ، تُعدل، وتصحح حلقة while الأقصر على شاشة واحدة- حيث ليست هناك حاجة لتجاوز الشاشة أو حدود الصفحة المطبوعة. ثالثًا، يستحق وضع الوظيفيتين ProcessDeposit() وProcessWithdrawal() في إجرائيات كل الحسنات الأخرى العامة للبنوية. رابعًا، الآن من السهل رؤية أنه بالإمكان فصل الشفرة إلى عبارة حالة، التي ستجعل الشفرة أسهل للقراءة، كما هو موضح تاليًا:

مثال بلغة البرمجة سي++ لشفرة مُتداخلة جيدة بعد التحليل الوظيفي واستخدام عبارة الحالة.

```
while ( !TransactionsComplete() ) {
// transaction قراءة سجل
 transaction = ReadTransaction();
 // اعتمادا على نوعه transaction معالجة
 switch ( transaction.Type ) {
 case ( TransactionType_Deposit ):
 ProcessDeposit(
 transaction.AccountType ،
 transaction.AccountSubType،
 transaction.AccountNum ،
 transaction.Amount
 );
 break;
```

```
case ( TransactionType_Withdrawal ):
ProcessWithdrawal(
transaction.AccountType ،
transaction.AccountNum،
transaction.Amount
);
break;
case ( TransactionType_Transfer ):
MakeFundsTransfer(
transaction.SourceAccountType،
transaction.TargetAccountType،
transaction.AccountNum ،
transaction.Amount
);
break;
default:
// معالجة الأنواع غير المعروفة
LogTransactionError("Unknown Transaction Type" ،transaction );
break;
}
}
```

استخدم الطريقة كائنية التوجه بشكل أكبر. إن الطريقة الأبسط لتبسيط هذه الشفرة بشكل خاص في بيئة كائنية التوجه، هي بإنشاء صف وصفوف فرعية لقاعدة إجراء مجردة، من أجل *Deposit* و*Withdrawal* و*Transfer*.

مثال بلغة البرمجة شي++ لشفرة جيدة تستخدم تقنية تعدد الأشكال.

```
TransactionData transactionData;
Transaction *transaction;
while ( !TransactionsComplete() ) {
 // اقرأ سجل المناقلة
 transactionData = ReadTransaction();
 // إنشاء كائن المناقلة، اعتمادا على نوع المناقلة
 switch ( transactionData.Type ) {
 case ( TransactionType_Deposit ):
 transaction = new Deposit( transactionData );
 break;
 case ( TransactionType_Withdrawal ):
```

```
 transaction = new Withdrawal( transactionData );
 break;
 case ( TransactionType_Transfer ):
 transaction = new Transfer( transactionData );
 break;
 default:
 // معالجة نوع مناقلة غير معروف
 LogTransactionError("Unknown Transaction Type" ،transactionData );
 return;
 }
 transaction->Complete();
 delete transaction;
}
```

في نظام من أي حجم، سيتم تحويل عبارة switch ليتم استخدام طريقة تقسيم الشفرة، حيث سيتم إعادة استخدام كائن من النوع Transaction في كل مكان يوجد فيه حاجة لإنشاء هذا النوع. إذا كانت هذه الشفرة في مثل هكذا نظام، سيكون هذا الجزء منها حتى أبسط[1]:

مثال بلغة البرمجة سي++ لشفرة جيدة تستخدام تقنية تعدد الأشكال وإنشاء كائن.

```
TransactionData transactionData;
Transaction *transaction;
while ( !TransactionsComplete() ) {
 // قراءة سجل المناقلة و إكمالها
 transactionData = ReadTransaction();
 transaction = TransactionFactory.Create( transactionData );
 transaction->Complete();
 delete transaction;
```

من أجل السجل، إن الشفرة في الإجرائية () TransactionFactory.Createهي تكيف بسيط للشفرة من المثال السابق عن استخدام عبارة switch:

مثال بلغة البرمجة سي++ عن شفرة جيدة لاستخدام كائن

```
Transaction *TransactionFactory::Create(
 TransactionData transactionData
 ) {
```

[1] **إشارة مرجعية**: للمزيد من تحسينات الشفرة المفيدة مثل هذه، انظر الفصل 24 " إعادة التصنيع".

```
إنشاء كائن transaction// اعتمادا على نوعه
switch ( transactionData.Type ) {
case ( TransactionType_Deposit ):
return new Deposit( transactionData );
break;

case ( TransactionType_Withdrawal ):
return new Withdrawal( transactionData );
break;

case ( TransactionType_Transfer ):
return new Transfer( transactionData );
break;

default:
// transaction معالجة الأنواع غير المعروفة لـ
LogTransactionError( "Unknown Transaction Type" ،transactionData );
return NULL;
}
}
```

أعد كتابة الشفرة المُتداخلة بشكل عميق. يجادل بعض الخبراء حول أنه تُشير عبارات الحالة افتراضيًا إلى شفرة مقسمة بشكل ضعيف في البرمجة غرضية التوجيه، وأن هذه العبارات الحادة نادرًا ما يكون هناك حاجة لاستخدامها (ماير 1997). إن هذه التحويل من عبارات الحالة الموجودة في الإجرائيات إلى استخدام كائن مع استدعاءات متعددة الأشكال هو مثال عن هذا.

أكثر عمومًا، إن الشفرة المعقدة هي علامة على عدم فهمك لبرنامجك بشكل كافي لجعله بسيط. إن التداخل العميق هو علامة تحذير تُشير إلى الحاجة إلى فصل الشفرة في إجرائية أو إلى إعادة تصميم الجزء من الشفرة المُعقد. هذا لا يعني أنه عليك تعديل الإجرائية، ولكن يجب أن تملك سبب جيد لعدم القيام بما لا ترغب القيام به.

مُلخص لتقنيات إنقاص عمق التداخل

فيما يلي لائحة للتقنيات التي يمكنك استخدامها لإنقاص عمق التداخل، مع المراجع لأقسام الكتاب التي تُناقش هذه التقنيات:

- إعادة اختبار جزء من الشرط (هذا القسم)
- التحويل إلى عبارات if-then-else (هذا القسم)
- التحويل إلى عبارة حالة (هذا القسم)
- فصل الشفرة المُتداخلة بشكل عميق في إجرائية خاصة بها (هذا القسم)
- استخدام الكائنات وإنجاز تقنية تعدد الأشكال (هذا القسم)
- إعادة كتابة الشفرة لاستخدام مُتغير الحالة (القسم 3.17)
- استخدام جُمل الحماية للخروج من إجرائية وجعل المسار الطبيعي خلال الشفرة أوضح (القسم 1.17)
- استخدام التعابير (القسم 4.8)
- إعادة تصميم الشفرة المُتداخلة بشكل عميق داخلياً (هذا القسم)

5.19 أساس برمجي: البرمجة الهيكلية

تم تقديم المُصطلح "البرمجة المُنظمة" لأول مرة في الورقة العلمية المعروفة "البرمجة المُنظمة" المُقدمة من قبل إدجر ديجكسترا عام 1969 في مؤتمر ناتو لهندسة البرمجيات (ديجكسترا 1969). بمرور الوقت جاء وذهب مُصطلح البرمجة المُنظمة، وبقي مُصطلح "مُنظمة"، حيث يتم تطبيقه في كل نشاط مُتعلق بتطوير البرمجيات، بما فيها التحليل المُنظم، التصميم المُنظم، والخطأ المُنظم. لن يجمع هذه المنهجيات المُنظمة المتنوعة أية موضوع مُشترك ماعدا أنه قد تم إنشاءهم جميعًا في وقت واحد عندما أعطتهم الكلمة "مُنظمة" دمغة إضافية.

إن جوهر البرمجة المُنظمة هو الفكرة البسيطة بأن على البرنامج أن يستخدم فقط بُنية تحكم واحدة للدخل وواحدة للخرج (أيضًا تسمى بنى تحكم دخل- وحيد، خرج-وحيد). إن بنى التحكم الدخل-الوحيد، الخرج-الوحيد، هي عبارة عن بلوك من شفرة لديه مكان وحيد فقط يمكنه البدء منه ومكان وحيد فقط يمكنه الإنهاء عنده. حيث ليس لهذا البلوك من الشفرة أية مداخل أو مخارج أخرى. إن البرمجة المُنظمة ليست مثل التصميم من الأعلى إلى الأسفل المُنظم. حيث يتم تطبيقه فقط على مستوى كتابة شفرة مُفصّل.

يسير البرنامج المُنظم بطريقة منضبطة بدلًا من القفز من مكان إلى أخر بشكل غير متوقع. يمكنك أن تقرأ هذا البرنامج من الأعلى إلى الأسفل، وينفذ بالطريقة نفسها. تؤدي المقاربات الأقل انضباطًا إلى شفرة مصدر توفر أقل صورة ذات معنى، صورة أقل قابلية للقراءة لكيفية تنفيذ البرنامج في الألة. تعني قابلية القراءة القليلة إلى فهم قليل، وبالنهاية إلى جودة برنامج أقل.

لا تزال المفاهيم المركزية للبرمجة المنظمة مُفيدة اليوم وتطبق على الاعتبارات باستخدام break، continue، throw، catch، ومواضيع أخرى.

المكونات الثلاثة للبرمجة المنظمة

تصف الأقسام القليلة التالية البنى الثلاث التي تُشكل جوهر البرمجة المنظمة.

السلسلة[1]

إن السلسلة هي مجموعة من العبارات البرمجية المُنفذة في ترتيب ما. تتضمن العبارات البرمجية المُتسلسلة بشكل نموذجي عمليات الإسناد واستدعاءات الإجرائيات.

مثال بلغة البرمجة جافا لشفرة متسلسلة.

```
// سلسلة من عبارات الاسناد
a = "1";
b = "2";
c = "3";
// سلسلة من استدعاء الإجرائيات
System.out.println( a );
System.out.println( b );
System.out.println( c );
```

الاختبار[2]

إن الاختبار هو بنية تحكم تؤدي إلى تنفيذ العبارات البرمجية بشكل انتقائي. إن عبارة if-then-else هي مثال شائع. حيث يتم تنفيذ إما بند if-then أو بند else، ولكن ليس كلاهما. أي يتم "اختيار" بند واحد من البنود للتنفيذ.

عبارة الحالة case هي مثال آخر عن التحكم الانتقائي. إن العبارة البرمجية switch في سي++ وجافا هي عبارة select في الفيجول بيسك، وهي كلها عبارة عن مثال عن عبارة الحالة case. في كل حالة، يتم تنفيذ حالة واحدة من حالات متعددة للتنفيذ. من الناحية النظرية إن عبارة if وعبارات الحالة هي عبارات مُتشابهة.

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول استخدام السلسة، انظر الفصل 14، "تنظيم الشفرة الخطية "

[2] **إشارة مرجعية**: لمزيد من التفاصيل حول استخدام الاختيار، انظر الفصل 15 "استخدام الشروط".

إذا لم تدعم لغتك البرمجية عبارات الحالة، فإنه بإمكانك أن تُحاكيهم باستخدام عبارات `if`. فيما يلي مثالين عن الاختيار:

مثال بلغة البرمجة جافا عن الاختيار.

```
//if تحديد بعبارة
if ( totalAmount > 0.0 ) {
 // افعل شيء ما
 ...
}
else {
 // افعل شيء ما آخر
 ...
}
//case التحديد في عبارة
switch ( commandShortcutLetter ) {
 case 'a':
 PrintAnnualReport();
 break;
 case 'q':
 PrintQuarterlyReport();
 break;
 case 's':
 PrintSummaryReport();
 break;
 default:
 DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

التكرار[1]

إن التكرار هو بنية تحكم يتم فيها تنفيذ مجموعة من العبارات البرمجية مرات متعددة. يتم الإشارة عادةً إلى مُصطلح التكرار باستخدام المُصطلح "حلقة". تتضمن أنواع التكرار `For-Next` في فيجول بيسك، و`while` و`for` في سي++ وجافا. يُظهر مقطع الشفرة هذا أمثلة عن التكرار في الفيجول بيسك:

أمثلة بلغة البرمجة فيجول بيسيك عن التكرار.

[1] **إشارة مرجعية:** لمزيد من التفاصيل عن استخدام التكرارات، انظر الفصل 16، "الحلقات المُتحكمة".

```
' example of iteration using a For loop
For index = first To last
 DoSomething( index )
Next
' example of iteration using a while loop
index = first
While ( index <= last )
 DoSomething ( index )
 index = index + 1
Wend
' example of iteration using a loop-with-exit loop
index = first
Do
 If ( index > last ) Then Exit Do
 DoSomething ( index )
 index = index + 1
Loop
```

إن جوهر البرمجة المُنظمة هو استخدام أية تدفق تحكم، يمكن أن يتم انشاءه باستخدام بنى التحكم الثلاثة من السلسة والاختيار والتكرار (بوم جاكو- بيني 1966). يُفضل المبرمجين في بعض الأحيان بنى لغة تزيد السهولة، ولكن يبدو أن البرمجة تقدمت إلى حد كبير من خلال تقييد ما يُسمح لنا باستخدامه بلغات البرمجة الخاصة بنا. قبل البرمجة المُنظمة، تم استخدام goto التي قدمت راحة قصوى في تدفق التحكم، ولكن الشفرة المكتوبة فيها لم تكن تملك قابلية الفهم وقابلية الصيانة. اعتقادي هو أن استخدام بنى التحكم غير بنى التحكم الثلاثة القياسية للبرمجة المنظمة-مثلا استخدام break، continue، return، throw-catch وإلى آخره – يجب توخي الحذر معه.

6.19 هياكل التحكم والتعقيد

إن السبب الوحيد الذي يؤدي إلى لفت الانتباه بشكل كبير إلى بنى التحكم، هو أن هذه البنى هي المُساهم الأساسي بزيادة تعقيد البرنامج الذي يستخدم هذه البنى. يزيد الاستخدام الضعيف لبنى التحكم من التعقيد؛ ويُنقص الاستخدام الجيد من التعقيد.

إن أحد مقاييس "التعقيد البرمجي" هو عدد العناصر التي يجب أن تُحافظ عليها في ذاكرتك من أجل فهم البرنامج.[1] هذا التفكير العقلي هو واحد من أصعب المفاهيم في البرمجة وهو السبب في أن البرمجة تتطلب

[1] اجعل الأشياء بسيطة قدر الإمكان- ولكن ليس أبسط- البرت اينشتاين

تركيز أكبر من النشاطات الأخرى. وهو سبب انزعاج المبرمجين من "المقاطعات السريعة"- مثل هكذا مقاطعات هو المحافظة على ثلاث كُرات في الهواء وحمل سلع قُمت بشرائها في نفس الوقت.



بشكل حدسي، يبدو أنه يحدد تعقيد برنامج إلى حد كبير كمية الجهد المطلوب لفهمه. قام توم مكابي بنشر ورقة مؤثرة مُناقشًا فيها فكرة أن تعقيد البرنامج مُحدد بواسطة تدفق التحكم (1976). حدد باحثين الأخرين عوامل أخرى غير التي حددها McCab (مثل عدد المتغيرات المُستخدمة في الإجرائية)، ولكنهم ناقشوا بأن تدفق التحكم هو على الأقل واحد من أكبر المُساهمات في التعقيد، إذا لم يكن الأكبر.

ما مدى أهمية التعقيد؟

قلق باحثين علوم الحاسوب حول أهمية التعقيد لمدة عقدين على الأقل.[1] منذ عدة سنوات، حذّر إيدجر ديجكسترا من مخاطر التعقيد: " إن المبرمج المُختص مُدرك بشكل كامل لحدود حجم جمجمته؛ ولهذا، فإنه يتعامل مع مهمة البرمجة بتواضع كامل (إيدجر ديجكسترا 1972) ". هذا لا يعني بأنه عليك أن تزيد سعة جمجمتك للتعامل مع التعقيد الضخم. بل هذا يعني بأنه لا يمكنك أبدًا التعامل مع التعقيد الضخم ويجب عليك أن تأخذ بعض الإجرائيات لإنقاص التعقيد حيثما أمكن.



تعقيد تدفق التحكم مُهم لأنه يتم تصحيحه بواسطة الموثوقية المنخفضة ومعدل الأخطاء المنخفض (مكابي 1976، شين وأخرين 1985). ذكر وليام وارد عن تحقيق مكاسب كبيرة في موثوقية البرامج الناتجة عن استخدام مقياس مكابي للتعقيد (ريك في هيوليت باكارد 1989). تم استخدام مقياس مكابي 1976 على برنامج من 77000 سطر لتعريف مناطق المشكلة. لدى البرنامج معدل عيب ما بعد النشر بقيمة 0.31 من العيوب لكل ألف سطر من الشفرة. حوى برنامج 125000 سطر على معدل عيب من 0.02 لكل ألف سطر من الشفرة. أعلن وارد عن هذا بسبب التعقيد المنخفض، لدى كلا البرنامجين عيوب أقل بكثير من البرامج الأخرى عند هيوليت باكارد. لدى شركتي الخاصة (Construx Software)، نفس النتائج باستخدام مقاييس التعقيد لتحديد الإجرائيات التي لديها مشكلة في الأعوام 2000.

الإرشادات التوجيهية العامة للحد من التعقيد

من الأفضل التعامل مع التعقيد باستخدام إحدى الطريقتين التاليتين. أولًا، يمكنك تحسين القدرات العقلية لديك باستخدام التمارين العقلية. ولكن البرمجة بحد ذاتها هي عبارة عن تدريب عقلي كافي، ويبدو أن لدى الناس

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول التعقيد، انظر "الحتمية التقنية الأساسية للبرمجيات: إدارة التعقيد " في القسم 2.5.

مشاكل عندما عليهم التفكير بخمس أو تسعة كائنات عقلية (ميلير 1956). أي إمكانات التحسين صغيرة. ثانيًا، يمكنك إنقاص تعقيد برامجك وكمية التركيز المطلوبة لفهمها.

كيفية قياس التعقيد

من الممكن أن يكون لديك شعور حدسي حول ما يجعل الإجرائية أكثر أو أقل تعقيدًا.[1] قد حاول الباحثين صياغة مشاعرهم الحدسية، وتوصلوا إلى عدة طرق لقياس التعقيد. من الممكن أن يكون أكثر التقنيات الرقمية المؤثرة هي تقنية مكوبي، التي فيها يتم قياس التعقيد بعدد "نقاط القرار" في إجرائية. يصف الجدول 19-2 طريقة عد نقاط القرار.

الجدول 19-2 تقنيات عد نقاط القرار في إجرائية

.1	ابدأ بالعدد 1 للمسار المستقيم في الإجرائية.
.2	أضف 1 لكل من الكلمات المفتاحية التالية، أو مُكافئاتها: if، while، repeat for، and، or .
.3	أضف 1 لكل حالة في عبارة الحالة case.

فيما يلي مثال عن ذلك:

```
if ( ( (status = Success) and done ) or
 ( not done and ( numLines >= maxLines ) ) ) then ...
```

في هذا المقطع، تبدأ العد بـ 1، ثم 2 من أجل if، ثم 3 من أجل and، ومن ثم 4 من أجل or، ومن ثم 5 من أجل and. وهكذا يحوي هذا المقطع على خمس نقاط قرار.

ماذا تفعل مع قياس التعقيد لديك

بعد الانتهاء من عد نقاط القرار، يمكنك استخدام هذا العدد لتحليل تعقيد الإجرائية:

0-5 الإجرائية ربما على ما يرام.

6-10 ابدأ بالتفكير في طرق لتبسيط الإجرائية.

+10 افصل جزء من الإجرائية في إجرائية ثانية واستدعيها من الإجرائية الاولى.

لا يُنقص نقل جزء من إجرائية إلى إجرائية أخرى من التعقيد الكلي للبرنامج؛ فقط يقوم بتحريك نقاط القرار. ولكن يُنقص هذا من كمية التعقيد التي يجب التعامل معها في وقت واحد. إن الهدف الأهم هو إنقاص عدد العناصر التي يجب عليك محاكمتها عقليًا، مع إنقاص جدير بالاهتمام لتعقيد الإجرائية المُعطاة.

[1] **قراءة متعمقة**: الطريقة الموصوفة هنا تعتمد على مقالة توم مكابي المهمة "قياس التعقيد"(1976).

ليس العدد الأعظمي (عشر نقاط قرار) بحد مُطلق. استخدم عدد نقاط القرار كعلم تحذير يُشير إلى الحاجة الممكنة لإعادة تصميم الإجرائية. لا تستخدم هذا كقاعدة غير مرنة. من الممكن أن تكون عبارة الحالة case ذات الحالات المتعددة أكثر من طول العناصر، بالاعتماد على الغرض من عبارة الحالة، من الممكن أن يكون من الحماقة كسر عبارة الحالة بعشر عبارات في هذه الحالة.

أنواع أخرى من التعقيد

إن مقياس مكابي McCabe للتعقيد ليس المقياس الوحيد،[1] ولكنه المقياس الأكثر مُناقشة في مراجع الحاسوب وهو بشكل أساسي مُفيد عندما تفكر ببنى التحكم. تتضمن المقاييس الأخرى كمية البيانات المُستخدمة، عدد مستويات التداخل في بنى التحكم، عدد السطور في الشفرة، عدد السطور بين المراجع الناجحة للمتغيرات ("امتداد")، عدد السطور التي فيها المُتغير مُستخدم ("زمن الحياة")، وكمية الدخل والخرج. طوّر بعض الباحثين مقاييس مُركبة تعتمد على تجميع مجموعة من هذه المقاييس الأبسط.

لائحة اختبار: قضايا التحكم العامة2

- هل تستخدم التعابير true وfalse بدلا من 1 و0؟
- هل يتم مقارنة القيم المنطقية مع true وfalse بشكل ضمني؟
- هل يتم مقارنة القيم العددية مع قيم اختباراتها بشكل صريح؟
- هل تم تبسيط التعابير بإضافة المتغيرات المنطقية واستخدام التوابع المنطقية وجداول القرار؟
- هل تم ذكر التعابير المنطقية بشكل إيجابي؟
- هل أزواج الأقواس متوازنة؟
- هل تم استخدام الأقواس في كل مكان تحتاجه من أجل التوضيح؟
- هل تم وضع الأقواس بشكل كامل للتعابير المنطقية؟
- هل تم كتابة الاختبارات في ترتيب رقم السطر؟
- هل تستخدم اختبارات جافا الأسلوب a.equals(b) بدلا من a == b عندما هناك إمكانية لذلك؟
- هل تم تجنب العبارات البرمجية الفارغة؟

[1] **قراءة متعمقة:** للمناقشة الممتازة حول مقاييس التعقيد، انظر مقاييس هندسة البرمجيات والنماذج ف (كونتي، ودونسمور ،وشن 1986)

[2] cc2e.com/1985

- هل تم تبســيط العبارات المُتداخلة بإعادة اختبار جزء من الشــرط، أو بتحويلها إلى if-then-else أو عبارات الحالة، أو بنقل الـشفرة المُتداخلة إلى داخل إجرائية خا صة بها، أو بالتحويل إلى التصميم المُعتمد على البرمجة غرضية التوجه، أو هل تم تحسينها بطريقة أخرى؟
- إذا كان لدى إجرائية رقم قرار أكبر من 10، هل يوجد ســبب جيد لعدم إعادة تصــميم هذه الإجرائية؟

نقاط مفتاحية

- يُساهم جعل التعابير المنطقية أبسط وأكثر قابلية على القراءة بشكل أكبر على جودة شفرتك.
- يجعل التداخل "التعشــيش" العميق الإجرائية أصــعب على الفهم. لحســن الحظ، يمكنك تجنب هذا بشكل سهل نسبيا.
- البرمجة الهيكلية هي فكرة بسيطة لا تزال ذات صلة: يمكنك بناء أي برنامج من مجموعة من السلاسل والاختيارات والحلقات.
- إن إنقاص التعقيد هو مفتاح الأساسي لكتابة شفرة عالية الجودة.

القسم الخامس: تحسينات الشفرة

في هذا القسم:

الفصل 20

المنظر الطبيعي لجودة البرمجيات

المحتويات[1]

مواضيع ذات صلة

- البناء التعاوني: الفصل 21
- اختبار المطور: الفصل 22
- التصحيح: الفصل 23
- متطلبات البناء: الفصل 3 و4
- هل تُطبّق المتطلبات على المشاريع البرمجية المعاصرة؟: القسم 3. 1

يستعرض هذا الفصل تقنيات "جودة البرمجيات" من وجهة نظر "البناء". بالطبع، يتحدث الكتاب كله عن تحسين جودة البرمجيات، لكن هذا الفصل يركز على الجودة وضمان الجودة بالضبط. يركز على قضايا "الصورة الكبيرة" أكثر من التقنيات العملية. إذا كنت تبحث عن نصيحة تطبيقية بخصوص التطوير والاختبار والتصحيح التعاوني، تقدم إلى الفصول الثلاثة التالية.

[1] cc2e.com/2036

1.20 مميزات جودة البرمجيات

تمتلك البرمجيات مميزات الجودة الداخلية والخارجية كليهما. المميزات الخارجية هي المميزات التي يهتم بها مستخدم المنتج البرمجي، وتتضمن التالي:

- **الصحة** درجة خلوّ النظام من الأخطاء في المواصفات والتصميم والتحقيق.
- **قابلية الاستخدام** الراحة التي يستطيع بها المستخدمون تعلّم واستخدام النظام
- **الفعّالية** الاستخدام الأصغري لموارد النظام، متضمنة الذاكرة ووقت التنفيذ.
- **الموثوقية** قدرة النظام على إنجاز الوظائف المطلوبة تحت الشروط المعلنة متى طُلبت-أن يكون لدينا وقت وسطي طويل بين الإخفاقات.
- **التكامل** درجة منع النظام للدخول غير الصحيح أو بلا إذن إلى برامجه وبياناته. فكرة التكامل تتضمن تقييد دخول المستخدم غير المصرّح له بالإضافة إلى ضمان الدخول إلى البيانات بطريقة صحيحة- هذا يعني، أنّ جداول البيانات التفرعيّة يتم تعديلها على التفرّع، وأن حقول التاريخ تحتوي تواريخ صالحة فقط، وإلى ما هنالك.
- **التكيّف** مدى إمكانية استخدام البرنامج، بدون تعديلات، في التطبيقات أو البيئات المغايرة لتلك التي صُمم خصيصاً لها.
- **الدقة** درجة خلوّ النظام، مذ بُني، من الأخطاء، خصوصاً فيما يتعلق بالمخرجات الكمّيّة. تختلف الدقة عن الصحة؛ إنها تحديد لمدى حُسن قيام النظام بعمله الذي بني من أجله بدلاً من إن كان قد بُني بشكل صحيح.
- **القوة** درجة استمرار النظام بوظيفته بحضور المدخلات غير الصالحة أو ظروف البيئة المسببة للتوتر.

يتداخل بعض هذه المميزات، لكن لكل واحدة ظلال مختلفة من المعاني التي تُطبّق أكثر في بعض الحالات، وأقل في الأخرى.

المميزات الخارجية للجودة هي النوع الوحيد من مميزات البرمجيات التي يهتم به المستخدمون. يهتم المستخدمون بكون البرمجيات سهلة الاستخدام أو لا، لا بكونها سهلة التعديل بالنسبة لك أو لا. يهتمون بكون البرمجية تعمل بشكل صحيح أو لا، لا بكون الشفرة مقروءة أو مُهيكلة جيداً أو لا.

يهتم المبرمجون بالمميزات الداخلية للبرمجيات بالإضافة إلى الخارجية. هذا الكتاب مركّز على الشفرة، لذا فهو يركز على مميزات الجودة الداخلية، متضمنة

- **قابلية الصيانة** درجة السهولة التي تستطيع بها أن تعدل نظام البرمجية لتغيّر أو تضيف مقدرات، أو تحسن أداء، أو تصحح أخطاء.

- **المرونة** المدى الذي إليه تستطيع أن تعدل نظاماً لاستخدامات أو بيئات أخرى غير التي صُمم خصيصاً لها.
- **المحمولية** درجة السـهولة التي تسـتطيع بها أن تعدل نظاماً ليعمل في بيئة مختلفة عن التي صُـمم خصيصاً لأجلها.
- **قابلية التدوير** المدى الذي إليه ودرجة السهولة التي بها تستطيع استخدام أجزاء من نظام في أنظمة أخرى.
- **قابلية القراءة** درجة السهولة التي تستطيع بها أن تقرأ وتفهم الشفرة المصدرية لنظام، خصو صاً في مستوى العبارة المفصّلة.
- **قابلية الاختبار** الدرجة التي إليها تستطيع القيام ب "اختبار الوحدة" و"اختبار النظام" لنظام؛ الدرجة التي إليها تستطيع التأكد أن النظام يلبّي المتطلبات.
- **قابلية الفهم** درجة السـهولة التي بها تسـتطيع فهم نظام في مسـتويي "تنظيم النظام" و"العبارة المفصلة". تتعلق "قابلية الفهم" بتماسك النظام في مستوٍ أكثر عمومية مما تفعل "قابلية القراءة".

كما في لائحة مميزات الجودة الخارجية، تتداخل بعض هذه المميزات الداخلية، لكن لها أيضاً ظلال مختلفة من المعاني القيّمة.

الجوانب الداخلية لجودة النظام هي الموضوع الرئيسي لهذا الكتاب ولم تُناقش أكثر في هذا الفصل.

الفرق بين المميزات الداخلية والخارجية ليس واضح تماماً لأن المميزات الداخلية في بعض المستويات تؤثر في الخارجية. تُضعف البرمجيات غير المفهومة أو غير القابلة للصيانة من الداخل من قدرتك على تصحيح العيوب، الذي بدوره يؤثر على المميزات الخارجية من الصحة وقابلية القراءة. لا يمكن تحسين البرمجيات غير المرنة لتتجاوب مع متطلبات المستخدم، الذي بدوره يؤثر على المميزات الخارجية من قابلية الاستخدام. الفكرة هنا أنه تم التأكيد على أن بعض مميزات الجودة تجعل الحياة أسهل على المستخدم، وبعضٌ آخر يجعلها أسهل على المبرمج. حاول أن تعرف إحداها من الأخرى ومتى وكيف تتفاعل هذه المميزات.

تتعارض محاولة تعظيم مميزات محددة حتماً مع محاولة تعظيم مميزات أخرى. إيجاد المحلول "الخلاصة" الأمثل من مجموعة أهداف متنافسة هو أحد النشاطات التي تجعل تطوير البرمجيات فرع هندسة حقيقي. يظهر الشكل 20-1 طريقة تأثير التركيز على بعض مميزات الجودة الخارجية بالمميزات الأخرى. يمكن أن توجد نفس الأنواع من العلاقات بين المميزات الداخلية لجودة البرمجيات.

الجانب الأكثر متعة في هذا الشكل أن التركيز على مميزة محددة لا يعني دائماً مقايضة بمميزة أخرى. أحياناً إحداها تؤذي أخرى، مثالاً، الصحة هي مميزة العمل بالضبط وفق المواصفات. القوة هي القدرة على الاستمرار بالعمل حتى تحت الشروط غير المتوقعة. يؤذي التركيز على الصحة القوة وبالعكس. خلافاً لذلك، يساعد التركيز على قابلية التكيّف القوة وبالعكس.

يظهر الشكل فقط العلاقات النمطية بين مميزات الجودة. في أي مشروع معطى، قد تمتلك مميزتين علاقة مختلفة عن العلاقة النمطية. من المفيد التفكير بأهداف جودة محددة خاصة بك وإن كان كل زوج من الأهداف بشكل متبادل نافع أو عدائي.



كيف يؤثر التركيز على العامل تحت بالعامل إلى اليسار	الصحة	قابلية الاستخدام	الفعالية	الموثوقية	التكامل	قابلية التكيّف	الدقة	القوة
الصحة	↑		↑	↑			↑	↓
قابلية الاستخدام		↑				↑	↑	
الفعالية	↓		↑	↓	↓	↓	↓	
الموثوقية	↑			↑	↑		↑	↓
التكامل			↓	↑	↑			
قابلية التكيّف					↓	↑		↑
الدقة	↑		↓	↑		↓	↑	↓
القوة	↓	↑	↓	↓	↓	↑	↓	↑

الشكل 20-1 يمكن أن يؤثر التركيز على واحدة من المميزات الخارجية لجودة البرمجيات بمميزات أخرى إيجابياً، أو سلبياً، أو لا يؤثر مطلقاً

20. 2 تقنيات لتطوير جودة البرمجيات

ضمان جودة البرمجيات هو برنامج مخطَّط ومنظَّم من النشاطات المُصممة لتأكيد امتلاك النظام للمميزات المطلوبة. قد تبدو الطريقة الفضلى لتطوير منتج عالي الجودة هي التركيز على المنتج نفسه، رغم ذلك، في ضمان جودة البرمجيات إنك تحتاج أيضاً أن تركز على عملية تطوير البرمجية. بعض عناصر برنامج جودة البرمجيات موصوف في الأقسام الفرعية التالية:

أهداف جودة البرمجيات إحدى التقنيات القوية لتحسين جودة البرمجية هي وضع أهداف جودة واضحة من بين المميزات الداخلية والخارجية الموصوفة في القسم السابق. بدون أهداف واضحة، قد يعمل المبرمجون لتعظيم مميزات مختلفة عن التي تتوقع أن يعظّموها. قوة وضع أهداف واضحة مُناقَشة بتفصيل أكثر لاحقاً في هذا القسم.

نشاط ضمان جودة واضح مشكلة شائعة في ضمان الجودة هي أنّ يُفهم أنّ الجودة هدف ثانوي. حقيقةً، في بعض المنظمات، البرمجة السريعة والوسخة هي القاعدة بدلاً من أن تكون الاستثناء. يُكافأ المبرمجون مثل "*غاري* العالمي"، الذين يوسّخون شفرتهم بالعيوب و "يتمّون" برامجهم بسرعة، أكثر من المبرمجين مثل "*هنري* عالي الجودة"، الذين يكتبون برامج ممتازة ويتأكدون أنها قابلة للاستخدام قبل إصدارها. في مثل هذه المنظمات، لا ينبغي أن يكون مفاجئاً ألا يجعل المبرمجون الجودة أوليتهم الأولى. يتوجب على المنظمة أن تبيّن للمبرمجين أن الجودة أولوية. جعلُ نشاط ضمان الجودة بيّناً يجعل الأولوية واضحة، وسيستجيب المبرمجون تبعاً.

استراتيجية الاختبار[1] يمكن أن يؤمن اختبار التنفيذ تخميناً مُفصَّلاً عن موثوقية المنتج. قسم من ضمان الجودة هو تطوير استراتيجية اختبار بالاقتران مع متطلبات وهيكلة وتصميم المنتج. يعتمد المطورون في العديد من المشاريع على الاختبار كطريقة رئيسية لكلا تخمين الجودة وتطوير الجودة. تشرح بقية هذا الفصل بتفصيل أكثر أنّ هذا حمل ثقيل جداً على الاختبار ليحمله لوحده.

توجيهات هندسة البرمجيات[2] ينبغي أن تسيطر التوجيهات على الميزة التقنية للبرمجية بينما تُطوَّر(البرمجية). تُطبَّق هكذا توجيهات على كل نشاطات تطوير البرمجية، متضمنة تعريف المشكلة وتطوير المتطلبات والهيكلة والبناء واختبار النظام. التوجيهات في هذا الكتاب هي، بأحد المعاني، مجموعة من توجيهات هندسة البرمجيات للبناء.

المراجعات التقنية غير الرسمية يراجع العديد من مطوري البرمجيات أعمالهم قبل تسليمها للمراجعة الرسمية. تتضمن المراجعات غير الرسمية "اختبار مكتبي" للتصميم أو للشفرة أو المرور على الشفرة مع بضعة نظراء.

المراجعات التقنية الرسمية[3] أحد أقسام إدارة عملية هندسة البرمجية أن تقبض على المشاكل في المرحلة "الأدنى قيمةً"-هذا يعني، في الوقت الذي يكون قد عُمل فيه أقل استثمار والذي تكلف فيه المشاكل أقل ما يمكن لتُصلح. لإنجاز هكذا هدف، يستخدم المبرمجون "بوابات الجودة،" (وهي) الاختبارات أو المراجعات الدورية التي تحدد إن كانت جودة المنتج في مرحلة ما كافية لتدعم التقدم إلى التالية. تُستخدم بوابات الجودة

[1] **إشارة مرجعية** لتفاصيل حول الاختبار، انظر الفصل 22، "اختبار المطور"

[2] **إشارة مرجعية** لنقاش حول أحد أصناف توجيهات هندسة البرمجيات المناسبة للبناء، انظر إلى القسم 4. 2، "أعراف برمجية"

[3] **إشارة مرجعية** نوقشت المراجعات والتفحصات في الفصل 21، "البناء التعاوني"

عادةً للانتقال بين تطوير المتطلبات والهيكلة، والهيكلة والبناء، والبناء واختبار النظام. يمكن أن تكون "البوابة" "تفتيش"، أو مراجعة نظير، أو مراجعة زبون، أو "تدقيق".

لا تعني "البوابة" حاجة الهيكلة أو المتطلبات لتكون 100 بالمئة كاملة أو مجمَّدة[1]؛ إنها تعني فعلاً أنك ستستخدم البوابة لتحدد إن كانت المتطلبات أو الهيكلة جيدة كفاية لتدعم التطوير في المراحل اللاحقة. "جيد كفاية" قد تعني أنك صمّمت ال 20 بالمئة الأكثر حرجاً من المتطلبات أو الهيكلة، أو قد تعني أنك حددت 95 بالمئة بتفصيل موجع- يعتمد أي طرف ينبغي أن تصّوب عليه من المجال على طبيعة مشروعك المحدد.

التدقيقات الخارجية التدقيق الخارجي هو نوع محدد من المراجعات التقنية المستخدمة لتحديد حالة المشروع أو جودة المنتج الذي يُطوَّر. يُحضر فريق تدقيق من خارج المنظمة ويكتبون تقارير نتائجهم إلى من فوّض بالتدقيق، عادةً الإدارة.

عملية التطوير

كل من العناصر المذكورة حتى الآن مرتبطة بشكل صريح بضمان جودة البرمجية وبشكل ضمني بعملية تطوير البرمجية[2]. تؤدي جهود التطوير التي تحوي نشاطات ضمان الجودة إلى برمجيات أفضل بالمقارنة مع تلك التي لا تحوي. تؤثر عمليات أخرى ليست بشكل صريح نشاطات ضمان جودة بجودة البرمجيات أيضاً.

إجرائيات التحكم بالتغيّر[3] واحدة من العوائق الكبيرة أمام إنجاز جودة البرمجية هي التغيرات الخارجة عن السيطرة. يمكن أن تؤدي تغيرات المتطلبات الخارجة عن السيطرة إلى إرباك التصميم أو كتابة الشفرة. يمكن أن تؤدي التغيرات الخارجة عن السيطرة في التصميم إلى شفرة لا تتوافق مع متطلباتها، أو تفكك في الشفرة، أو وقت يُصرف في تعديل الشفرة لتوافق التصميم المتغيّر أكثر من الوقت الذي يُصرف لتحريك المشروع إلى الأمام. يمكن أن تؤدي التغيرات الخارجة عن السيطرة في الشفرة نفسها إلى تفكك داخلي وارتياب بشأن أي شفرة تمت مراجعتها واختبارها بشكل كامل وأيها لا. الأثر الطبيعي للتغيير هو زعزعة وتخفيض الجودة، لذا فإن التعامل مع التغيرات بفعالية هو المفتاح لإنجاز مستويات جودة عالية.

قياس النتائج مالم تقاس نتائج مخطط ضمان الجودة، لن تمتلك أية طريقة لمعرفة إن كان المخطط يعمل. يخبرك القياس إن كان المخطط ناجح أو فاشل ويسمح لك بتغيير عمليتك بطريقة خاضعة لسيطرتك لترى كيف

---

[1] **إشارة مرجعية** لتفاصيل أكثر عن كيفية اختلاف نُهج التطوير بالاعتماد على نوع المشروع، انظر القسم 3. 2، "تحديد نوع البرمجية التي تعمل عليها."

[2] **اقرأ أيضاً** لنقاش حول تطوير البرمجيات كعملية، انظر *تطوير البرمجيات الاحترافي* (ماكونيل 1994).

[3] **إشارة مرجعية** لتفاصيل عن تغيير التحكم، انظر القسم 28. 2 "إدارة الإعدادات"

يمكن أن يُطوَّر. تستطيع أيضاً قياس خصائص الجودة نفسهن-الصحة وقابلية الاستخدام والفعالية الخ-ومن المفيد أن تفعل ذلك. لتفاصيل عن قياس خصائص الجودة، انظر الفصل 9 من *Principles of Software Engineering* (غيلب 1988).

إعطاء النماذج الأولية "إعطاء النماذج الأولية" هو تطوير نماذج حقيقية لوظائف النظام المفتاحية. يستطيع المطور إعطاء نماذج أولية لأقسام من واجهة المستخدم لتحديد قابلية الاستخدام، أو لحسابات حرجة لتحديد وقت التنفيذ، أو لمجموعات بيانات قياسية لتحديد متطلبات الذاكرة. قارنَ فحص لأبحاث مفصلة منها 16 منشور و8 غير منشور "إعطاء النماذج الأولية" بمناهج تطوير المواصفات التقليدية. أظهرت المقارنة أن إعطاء النماذج الأولية يمكن أن يؤدي إلى تصاميم أفضل، ومطابقات أفضل لحاجات المستخدم، وقابلية صيانة محسَّنة (غوردون وبايمان 1991)

وضع الأهداف

إن وضع أهداف جودة بطريقة واضحة هو خطوة جليّة وبسيطة في إنجاز برمجيات ممتازة، لكن من السهل أن تُهمل، قد تتساءل إن وضعتَ أهداف جودة بيّنة، إذا كان المبرمجون سيعملون فعلاً لإنجازها؟ الجواب هو، نعم، سيفعلون، إذا علموا ما هي الأهداف وأن الأهداف منطقية. لا يستطيع المبرمجون الاستجابة لمجموعة أهداف تتغير يومياً أو مستحيلة التحقيق.

أشرف جيرلاد وينبرغ وإيدوارد اسكولمان على تجربة مثيرة لتحرّي الأثر على أداء المبرمجين بوضع أهداف جودة (1974). كان لديهم خمسة فرق من المبرمجين يعملون على خمسة إصدارات لنفس البرنامج. أُعطيت أهداف الجودة الخَمس نفسها لكل واحد من الفرق الخمس. وأُخبر كل فريق أن يحسّن هدف مختلف. أُخبر أحد الفرق أن يصغّر الذاكرة المطلوبة، وأُخبر آخر أن ينتج أوضح خرج ممكن، وأُخبر آخر أن يبني الشفرة الأكثر قابلية للقراءة، وأُخبر آخر أن يستخدم أقل عدد من العبارات، والمجموعة الأخيرة أُخبرت أن تكمل البرنامج بأقل مقدار ممكن من الزمن. يظهر الجدول 20-1 كيف ترتّب كل فريق بالنسبة إلى كل هدف.

جدول 20-1 ترتيب الفرق على كل هدف

الهدف الذي أُخبر الفريق أن يحسنه	استخدام ذاكرة أصغري	أكثر خرج قابلية للقراءة	أكثر شفرة قابلية للقراءة	أقل شفرة	وقت برمجة أصغري
الذاكرة الأصغرية	1	4	4	2	5
قابلية قراءة الخرج	5	1	1	5	3
قابلية قراءة البرنامج	3	2	2	3	4
الشفرة الأقل	2	5	3	1	3
وقت برمجة أصغري	4	3	5	4	1

المصدر: بتصرف من الأهداف والأداء في برمجة الحاسوب"Goals and Performance in Computer Programming" (وينبرغ وسكولمان)

بيانات مثبتة

نتائج هذه الدراسة كانت مميزة. أربعة من الفرق انتهوا بالمركز الأول في الهدف الذي أخبروا أن يحسنوه. الفريق الآخر انتهى بالمركز الثاني في هدفه. ولا واحد من الفرق طوّر بشكل متماسك وجيد لكل الأهداف.

الانطباع المفاجئ أن الناس يقومون فعلاً بما تطلب منهم أن يقوموا به. لدى المبرمجين حافز إنجاز عالي: سيعملون للوصول إلى الأهداف المحددة، لكن يتوجب أن يُخبروا ما هي الأهداف. الانطباع الثاني هو أن، كما هو متوقع، الأهداف تتعارض وبشكل عام من غير الممكن أن تعمل بشكل جيد من أجلها كلها.

3 .20 الفعالية النسبية لتقنيات الجودة

لا تمتلك كل التطبيقات المختلفة لضمان الجودة نفس الفعالية. دُرست تقنيات عديدة، وفعاليتها في اكتشاف وإزالة العيوب معروفة. يُناقش هذا وجوانب عديدة أخرى من "فعالية" تطبيقات ضمان الجودة في هذا القسم.

النسبة المئوية لاكتشاف العيوب

بعض التطبيقات أفضل في اكتشاف الأخطاء من الأخرى، والطرق المختلفة تجد أنواعاً مختلفة من العيوب[1]. إحدى طرق تقييم مناهج اكتشاف العيوب أن تحدد النسبة المئوية للعيوب التي تكتشفها من مجموع العيوب التي توجد في تلك اللحظة في المشروع. يظهر الجدول 20-2 النسب المئوية للعيوب المكتشفة بعدة تقنيات شائعة لاكتشاف الأخطاء.

جدول 20-2 معدلات اكتشاف العيوب

خطوة الإزالة	المعدل الأدنى	المعدل الطبيعي	المعدل الأعلى
مراجعات التصميم غير الرسمية	%25	%35	%40
تفحصات التصميم الرسمية	%45	%55	%65
مراجعات الشفرة غير الرسمية	%20	%25	%35
تفحصات الشفرة الرسمية	%45	%60	%70
النمذجة أو التصميمات البدئية	%35	%65	%80
الفحص المكتبي الشخصي للشفرة	%20	%40	%60
اختبار الوحدة	%15	%30	%50
اختبار الوظيفة (المكون) الجديدة	%20	%30	%35
اختبار التكامل	%25	%35	%40

[1] إذا بنى البناؤون الأبنية بالطريقة التي يكتب المبرمجون البرامج، عندها سيدمر المستعمرة أول نقار خشب يصل. —جيرالد وينبيرغ

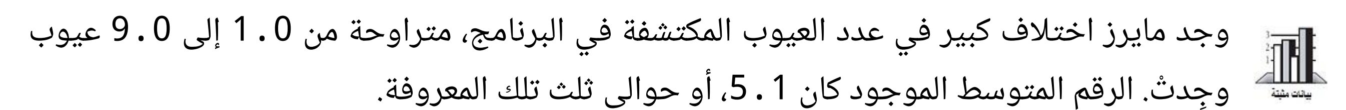
اختبار التراجع	%15	%25	%30
اختبار النظام	%25	%40	%55
اختبار بيتا منخفض الكمية (< 10 مواقع)	%25	%35	%40
اختبار بيتا عالي الكمية (< 1000 موقع)	%60	%75	%85

المصدر: بتصرف من Programming Productivity (جونز 1985)، "Software Defect-Removal Efficiency" (جونز 1996)، و "What We Have Learned About fighting Defects" (شل وآخرون 2002)

الحقائق الأكثر متعة التي تظهرها هذه البيانات هي أن المعدلات الطبيعية لا تتجاوز 75 بالمئة لأي تقنية مفردة، ومتوسط التقنيات حوالي 40 بالمئة. بعد، المعدلات الطبيعية لأكثر الأنواع شيوعاً من اكتشاف العيوب-اختبار الوحدة واختبار التكامل- هي فقط 30-35 بالمئة. تستخدم المنظمات النمطية نهج "مزيل عيوب" و"شديد الفحص" لتنجز فعالية إزالة العيوب فقط بحوالي 85 بالمئة. تستخدم المنظمات الرائدة تشكيلة أوسع من التقنيات وتنجز فعاليات إزالة عيوب بنسبة 95 بالمئة أو أعلى (جونز 2000).

التضمين القوي هو إن كان مطورو المشروع يكافحون من أجل معدل اكتشاف عيوب أعلى، فإنهم سيلزمون باستخدام تجميعة تقنيات. أكدت دراسة قديمة من قبل غلينفورد مايرز هذا التضمين (1978b). درس مايرز مجموعة مبرمجين بحد أدنى 7 ومتوسط 11 سنة من الخبرة المهنية. مستخدماً برنامج ب 15 خطأ معروف، طلب من كل مبرمج البحث عن الأخطاء باستخدام واحدة من هذه التقنيات:

- اختبار التنفيذ بالنسبة للمواصفات
- اختبار التنفيذ بالنسبة للمواصفات مع الشفرة المصدرية
- المرور-عبر/التفحص مستخدماً المواصفات والشفرة المصدرية



وجد مايرز اختلاف كبير في عدد العيوب المكتشفة في البرنامج، متراوحة من 1.0 إلى 9.0 عيوب وجِدث. الرقم المتوسط الموجود كان 5.1، أو حوالي ثلث تلك المعروفة.

عندما استُخدمت بشكل فردي، لم يكن لأي طريقة تقدم ملحوظ إحصائياً على أي من الطرق الأخرى. الاختلاف في الأخطاء التي وجدها الناس كان عظيماً جداً، على أي حال، حيث أن أي تجميعة من طريقتين-متضمنة وجود مجموعتين مستقلتين تستخدمان نفس المنهجية-زادت العدد الكلي للعيوب الموجودة بعامل يساوي تقريباً 2. أصدرت دراسات في **مختبر ناسا لهندسة البرمجيات**، و**بوينغ**، وشركات أخرى تقاريراً بأن أناس مختلفين يميلون لإيجاد عيوب مختلفة. فقط حوالي 20 بالمئة من مجموع الأخطاء الموجودة بالتفحص وجدها أكثر من متفحص (كوكاكديان، وغرين، وباسيلي 1989؛ ترب، وسترك، وبفلغ 1991؛ شنايدر، ومارتن، وتساي 1992).

أشار غرينفورد مايرز إلى أن العمليات الإنسانية (التفحصات والعبورات، على سبيل المثال) تميل لتكون أفضل من اختبار معتمد على الحاسوب في إيجاد أنواع محددة من الأخطاء والعكس صحيح لأنواع أخرى من الأخطاء (1979). أكدت هذه النتيجة في دراسة لاحقة، وجدت أن قراءة الشفرة تكتشف عيوب واجهات أكثر واختبار الوظائف يكتشف عيوب تحكم أكثر (باسيلي، وسيلبي، وهاتشن 1986). بلّغ الزعيم الروحي للاختبار بوريس بيزر أنّ نُهج الاختبار الرسمية عادةً تنجز تغطية للاختبار بنسبة 50-60 بالمئة فقط مالم تكن تستخدم محلل تغطية (جونسون 1994).



الثمرة هي أن منهجيات اكتشاف العيوب تعمل في تجميعات أفضل مما تعمل مفردة. قرر جونز نفس النقطة عندما لاحظ أن فاعلية اكتشاف العيوب التراكمية أكبر بشكل ملحوظ منها لأي تقنية مفردة. إن المنظر الخارجي لفعالية الاختبار المستخدم لوحده كالح. أشار جونز إلى أن تجميعة من اختبار الوحدة، واختبار الوظائف، واختبار النظام غالباً تنتج اكتشاف عيوب تراكمي أقل من 60 بالمئة، وهو غير ملائم لبرمجية إنتاجية.

يمكن أيضاً ان تستخدم هذه البيانات لفهم لمَ يختبر المبرمجون الذين يبدؤون العمل بتقنية إزالة عيوب منضبطة مثل **البرمجة الزائدة** مستويات إزالة عيوب أعلى من التي اختبروها من قبل. كما يوضح الجدول 20-3، يُتوقع لمجموعة من تطبيقات إزالة العيوب المستخدمة في **البرمجة الزائدة** أن تنجز فاعلية إزالة عيوب بنسبة حوالي 90 بالمئة بالحالة المتوسطة و97 بالمئة بأفضل الحالات، والتي هي بكثير أفضل من متوسط الصناعة لإزالة العيوب وهو 85 بالمئة. على الرغم من أن بعض الناس يربطون هذه الفعالية بالتعاون بين تطبيقات **البرمجة الزائدة**، فهي نتيجة متوقعة فقط لاستخدام هذه التطبيقات المحددة لإزالة العيوب. يمكن أن تعمل تجميعات أخرى من التطبيقات بجودة مساوية أو أفضل، وتقرير أية التطبيقات المحددة لإزالة العيوب لتستخدم كي تنجز مستوى الجودة المرغوب هو أحد أجزاء تخطيط المشروع الفعال.

جدول 20-3 معدل اكتشاف العيوب المُقدَّر للبرمجة الزائدة

خطوة الإزالة	المعدل الأدنى	المعدل الطبيعي	المعدل الأعلى
مراجعات التصميم غير الرسمية (برمجة الأزواج)	%25	%35	%40
مراجعات الشفرة غير الرسمية (برمجة الأزواج)	%20	%25	%35
الفحص المكتبي الشخصي للشفرة	%20	%40	%60
اختبار الوحدة	%15	%30	%50
اختبار التكامل	%25	%35	%40
اختبار التراجع	%15	%25	%30
فعالية إزالة العيوب التراكمية المتوقعة	%74~	%90~	%97~

كلفة إيجاد العيوب

تكلف بعض تطبيقات اكتشاف العيوب أكثر من بعضها الآخر. تؤدي التطبيقات الأكثر اقتصاد إلى أقل كلفة لكل عيب يُكتشف، في حال كانت كل الأمور الأخرى متساوية. شرط أنّ كل الأمور الأخرى يجب حتماً أن تكون متساوية مهم لأن كلفة كل عيب متأثرة بمجموع العيوب المكتشفة، والمرحلة التي اكتشف فيها كل عيب، والعوامل الأخرى بجانب الأمور الاقتصادية لتقنية اكتشاف العيوب المحددة.

بيانات مثبتة

وجدت معظم الدراسات أن التفحصات أرخص من الاختبارات. وجدت دراسة في **مختبر هندسة البرمجيات** أن قراءة الشفرة وجدت حوالي 80 بالمئة أخطاء في الساعة أكثر من الاختبار (باسيلي وسيلبي 1987). وجدت منظمة أخرى أن اكتشاف عيوب التصميم باستخدام الاختبار يكلف ست مرات أكثر منه باستخدام التفحصات (أكرمان، وبوتشواد، وليوسكي 1989). وجدت دراسة لاحقة في **أي بي إم** أنه لزمت فقط 3.5 ساعات عمل لإيجاد كل خطأ باستخدام تفحصات الشفرة، بينما لزمت 15-25 ساعة لإيجاد كل خطأ عن طريق الاختبار (كابلان 1995).

كلفة إصلاح العيوب

كلفة إيجاد العيوب هي أحد أقسام معادلة الكلفة. القسم الآخر هو كلفة إصلاح العيوب، قد يبدو للوهلة الأولى أن طريقة اكتشاف العيوب لا تهم-ستتكلف دائماً نفس المقدار لتصلّح.

هذا ليس صحيحاً لأنه كلما طال بقاء عيب في النظام[1]، أصبحت إزالته أكثر كلفة. لذا، تؤدي تقنية الاكتشاف التي تجد الأخطاء أبكر إلى كلف أقل لإصلاحها. وأكثر أهميةً حتى، بعض التقنيات مثل التفحصات، تكتشف الأعراض والمسببات للعيوب في خطوة واحدة؛ الأخرى، مثل الاختبار، تجد الأعراض لكن تتطلب عمل إضافي لتشخّص وتصلح العلّة الأساسية. النتيجة هي أن تقنيات الخطوة الواحدة هي فعلياً أقل كلفة من كل التقنيات ثنائية الخطوة.

بيانات مثبتة

وجدت شعبة التطبيقات في **مايكروسوفت** أن اكتشاف وإصلاح عيب ما يستغرق ثلاث ساعات باستخدام تفحصات الشفرة، و12 ساعة باستخدام الاختبار، التقنية ثنائية الخطوة (مور 1992). كتب كولوفيلو ووودفيلد تقارير على برنامج ب 700000 سطر مبنيّ من قبل 400 مطور (1989). لقد وجدوا أن فعالية الكلفة لمراجعات الشفرة كانت أكبر من فعالية الكلفة للاختبار بعدة مرات -عائد 1.38 على الاستثمار مقابل 0.17.

[1] **إشارة مرجعية** لتفاصيل حول حقيقة أن العيوب تصبح أثمن كلما بقيت أكثر في النظام، انظر "الطعن في البيانات" في القسم 3.1. لنظرة قريبة جداً على الأخطاء نفسها، انظر القسم 22.4، "الأخطاء النمطية"

الخلاصة هي أنه يتوجب على برنامج جودة البرمجيات الفعال أن يتضمن تجميعة تقنيات تُطبَّق على كل مراحل التطوير. إليك تجميعة مُزكّاة لإنجاز جودة أعلى من المتوسط:

- تفحصات رسمية لكل المتطلبات، وكل الهيكلة، وتصاميم الأجزاء الحساسة من النظام
- النمذجة أو إعطاء النماذج الأولية
- قراءة الشفرة أو التفحصات
- اختبار التنفيذ

4.20 متى تقوم بضمان الجودة

كما دوّن الفصل 3 ("قس مرتين، اقطع مرة: المتطلبات الأولية التحضيرية")،[1] كلما حُشر الخطأ في البرمجية أبكر، أصبحت الأمور أكثر تعقيداً في الأقسام الأخرى من البرمجية وأصبح الأمر مكلفاً أكثر لإزالته. يمكن أن يؤدي خلل في المتطلبات إلى خلل موافق واحد أو أكثر في التصميم، والذي يمكن أن يؤدي إلى الكثير من الخِلال الموافقة في الشفرة. يمكن أن يؤدي خطأ في المتطلبات إلى هيكلة زائدة أو إلى قرارات هيكلة سيئة. الهيكلة الزائدة تؤدي إلى شفرة وحالات اختبار وتوثيق زائدة. أو يمكن أن يؤدي خطأ في المتطلبات إلى هيكلة وشفرة وحالات اختبار منبوذة. تماماً كحُسْن فكرة أن تجد العيوب في الطبعة الزرقاء لبيت قبل أن تصب الإسمنت على الأساسات، حسنةٌ فكرة أن تقبض على أخطاء المتطلبات والهيكلة قبل أن تؤثر على النشاطات اللاحقة.

بالإضافة إلى أنه، تميل الأخطاء في المتطلبات أو الهيكلة لتكون أكثر امتداداً من أخطاء البناء. يمكن أن يؤثر خطأ هيكلة واحد على عدة صفوف ودزينات من الإجرائيات، بينما من غير المرجح أن يؤثر خطأ بناء واحد على أكثر من صف واحد أو إجرائية واحدة. لهذا السبب، أيضاً، إنه لأمر مؤثر بالكلفة أن تقبض على الأخطاء بالإبكار الذي تستطيع.



تنسلُّ العيوب إلى داخل البرمجية في كل المراحل. إذن، ينبغي أن تؤكد على العمل بضمان الجودة في المراحل المبكرة وخلال بقية المشروع. ينبغي أن يُخطّط له في المشروع عندما يبدأ العمل؛ وينبغي أن يكون جزء من الخيط التقني للمشروع بينما العمل يستمر؛ وينبغي أن يضع النقاط على الحروف في نهاية المشروع، مؤكِّداً صحة جودة المشروع عندما ينتهي العمل.

---

[1] **إشارة مرجعية** ضمان الجودة للنشاطات التحضيرية-المتطلبات والهيكلة، على سبيل المثال-خارج مجال هذا الكتاب. قسم "المصادر الإضافية" في نهاية الفصل يصف كتب بإمكانك أن تتحول إليها لمعلومات أكثر عن تلك النشاطات.

لغة عربية من لسان العرب جمع خلل خلال، مثل جبل وجبال. ومعنى الخلل هو الوهن والفساد.

لغة إنكليزية الطبعة الزرقاء لبيت هي مخطط البيت

5.20 المبدأ العام لجودة البرمجيات



لا يوجد شيء كغداء مجاني، وحتى إن وجد، لا توجد أية ضمانة أنه سيكون جيداً بأي شكل. تطوير البرمجيات بعيد جداً عن المطبخ الفرنسي، على أي حال، وجودة البرمجيات استثنائية بشكل ساطع. المبدأ الأساسي لجودة البرمجيات هو تحسين الجودة يخفّض كُلف التطوير.

يعتمد فهم هذا المبدأ على فهم المشاهدة المفتاحية: أفضل طريقة لتطوير الإنتاجية والجودة هو أن تخفّض الوقت المصروف على إعادة العمل في الشفرة، سواء أكانت إعادة العمل ناتجة عن تغيّرات في المتطلبات، أو تغيرات في التصميم، أو التصحيح. إنتاجية متوسط الصناعة في منتج برمجي هي حوالي 10 إلى 50 سطر من الشفرة المُسلَّمة من كل شخص في كل يوم (متضمنة كل الأعباء المغايرة لكتابة الشفرة). تستغرق كتابة 10 إلى 50 سطر من الشفرة مسألة دقائق فقط، لذا كيف تمر بقية اليوم؟

جزء من السبب وراء قيم الإنتاجية هذه المتدنية بشكل ظاهر هو أن قيم متوسط صناعة مثل هذه تُدرِج وقت غير المبرمج في رمز "أسطر الشفرة باليوم"[1]. وقت المختبر، ووقت مدير المشروع، ووقت الدعم الإداري كلها متضمنة. الأنشطة المغايرة لكتابة الشفرة، مثل تطوير المتطلبات وعمل الهيكلة، أيضاً تُدرَج عادةً في رموز "أسطر الشفرة باليوم" هذه. لكن ولا واحد منها هو ما يستهلك وقتاً كثيراً جداً.

النشاط الأكبر المتفرد في معظم المشاريع هو "التصحيح" وتصحيح الشفرة التي لا تعمل بشكل مناسب. التصحيح وإعادة التصنيع المرتبطة(به) وإعادة الأعمال الأخرى تستهلك حوالي 50 بالمئة من الوقت في دورة تطوير برمجية غِرّة تقليدية. (انظر القسم 3. 1، "أهمية المتطلبات،" لتفاصيل أكثر.) تقليل التصحيح بمنع الأخطاء يطوِّر الإنتاجية. لذا، معظم المناهج الظاهرة لتقصير جدول مواعيد التطوير هو تحسين جودة المنتج وتخفيض كمية الوقت المصروف على التصحيح وإعادة العمل في البرمجية.

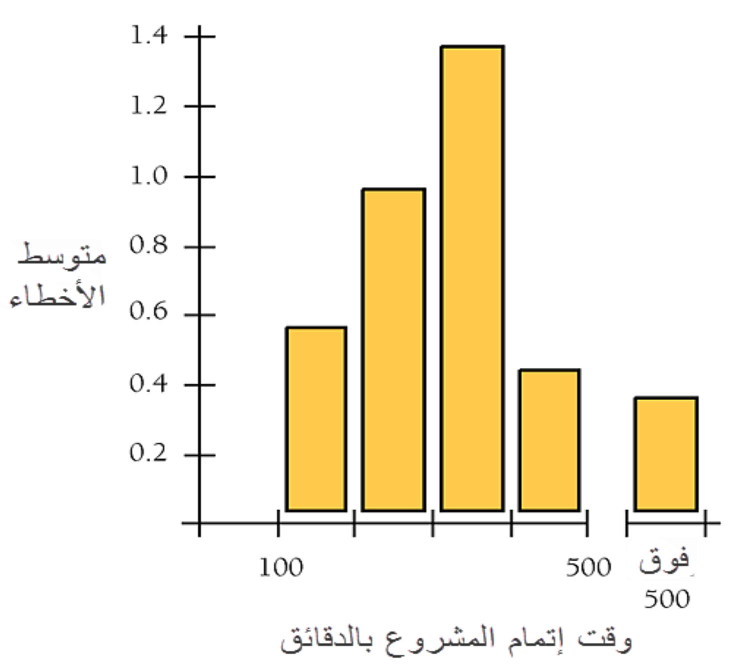
أكّد هذا التحليل ببيانات شاسعة. في مراجعة ل 50 مشروع تطوير تضمنت أكثر من 400 سنة عمل من الجهد وتقريباً 3 مليون سطر من الشفرة، وجدت دراسة في مختبر هندسة البرمجيات في *ناسا* أن ضمان الجودة المتزايد كان مرتبطاً بمعدل خطأ متناقص لكنه لم يزد كلفة التطوير الكلية (كارد 1987).

قدمت دراسة في IBM نتائج مشابهة:

[1] **إشارة مرجعية** لتفاصيل عن الفرق بين كتابة برنامج فردي وكتابة منتج برمجي، انظر "برامج ومنتجات وأنظمة ومنتج نظام" في القسم 27. 5.

تمتلك المشاريع البرمجية ذات المستويات الدنيا من العيوب جداول مواعيد التطوير الأقصر وإنتاجية التطوير الأعلى.... إزالة عيوب البرمجية هو فعلياً صيغة العمل الأكثر ثمناً واستهلاكاً للوقت في البرمجية (جونز 2000)

يبقى نفس الأثر صحيحاً في الطرف الصغير للمجال. في دراسة عام 1985، كتب 166 مبرمج محترف برامج من نفس المواصفات. تراوحت البرامج الناتجة حول 220 سطر من الشفرة وحول أقل بقليل من خمس ساعات كتابة. كانت النتيجة المدهشة أن المبرمجين الذين أخذوا الوقت الوسطي لإتمام برامجهم قدموا برامج بالعدد الأعظم من الأخطاء. المبرمجون الذين أخذوا أكثر أو أقل من الوقت الوسطي قدموا برامج بأخطاء أقل بشكل ملحوظ (ديماركو وليستير 1985). يرسم الشكل 20-2 النتائج بيانياً



الشكل 20-2 لا ينتج نهج التطوير الأسرع ولا الأبطأ البرمجية ذات العيوب الأكثر.

استغرقت المجموعتان الأبطأ حوالي خمس مرات من وقت المجموعة الأسرع لتنجز تقريباً نفس معدلها للعيوب. ليست حتميةً الحالةُ التي فيها تأخذ كتابة برمجية بدون عيوب وقتاً أكثر من كتابة برمجية بعيوب. كما يظهر الشكل البياني، يمكن أن تأخذ أقل.

بشكل لا يمكن إنكاره، في أنواع محددة من المشاريع، يكلف ضمان الجودة مالاً. إذا كنت تكتب شفرة لمكوك فضائي أو لنظام طبي لدعم الحياة، درجة الموثوقية المطلوبة تجعل المشروع أثمن.

بمقارنته مع دورة "كتابة الشفرة-الاختبار-التصحيح" التقليدية، يوفر برنامج جودة البرمجية المُنَّور مالاً. إنه يعيد توزيع الموارد بأخذها من التصحيح وإعادة التصنيع ووضعها في نشاطات ضمان الجودة التحضيرية.

النشاطات التحضيرية لها تأثير على جودة المنتج أكثر من النشاطات النهائية، لذا الوقت الذي تستثمره في التحضير يوفر وقتاً أكثر عند النهاية. الأثر الصافي هو عيوب أقل، ووقت تطوير أقصر، وكُلف أدنى. سترى عدة أمثلة عن **المبدأ العام لجودة البرمجية** في الفصول الثلاث التالية.

لائحة اختبار: مخطط لضمان الجودة

- هل عرّفت مميزات الجودة المحددة المهمة لمشروعك؟
- هل أعلمت الآخرين بأهداف جودة المشروع؟
- هل فكرت بالطرق التي فيها قد تنافس بعض المميزات أو تكامل المميزات الأخرى؟
- هل يستدعي مشروعك استخدام عدة تقنيات اكتشاف أخطاء مختلفة ملائمة لإيجاد عدة أنواع مختلفة من الأخطاء؟
- هل يتضمن مشروعك مخطط لاتخاذ خطوات لضمان جودة البرمجية خلال كل مرحلة في تطوير البرمجية؟
- هل تُقاس الجودة بطريقة تُمكّنك أن تخبر إن كانت تتحسن أم تسوء؟
- هل تفهم الإدارة أن ضمان الجودة يجلب كُلف في البداية لكي يوفر كُلف لاحقاً؟

مصادر إضافية

ليس من الصعب أن نذكر كتباً في هذا القسم لأنه فعلياً أي كتاب في منهجيات البرمجيات الفعالة يصف تقنيات تُنتج جودة وإنتاجية مُطورة.[1] الصعوبة هي إيجاد كتب تتعامل مع جودة البرمجية بحد ذاتها. هنا اثنان:

Ginac, Frank P. Customer Oriented Software Quality Assurance. Englewood Cliffs, NJ: Prentice Hall, 1998. هذا كتاب قصير جداً يصف واصفات الجودة، وواحداتها، وبرامج ضمان الجودة، والقاعدة في اختبار الجودة، بالإضافة إلى برامج تحسين الجودة المعروفة جيداً، متضمنة "نموذج إدراك المقدرات" التابع لجمعية هندسة البرمجيات وآيزو9000.

Lewis, William E. Software Testing and Continuous Quality Improvement, 2d ed. Auerbach Publishing, 2000. يقدم هذا الكتاب نقاش مسهب حول دورة حياة الجودة، بالإضافة إلى نقاش موسع حول تقنيات الاختبار. ويقدم أيضاً صيغ ولوائح متعددة.

معايير ذات صلة

IEEE Std 730-2002, IEEE Standard for Software Quality Assurance Plans.[2]
معيار لمخططات ضمان جودة البرمجيات

IEEE Std 1061-1998, IEEE Standard for a Software Quality Metrics Methodology.
معيار لمنهجيات قياسات جودة البرمجية

IEEE Std 1028-1997, Standard for Software Reviews.
معيار لمراجعات البرمجية

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing.
معيار لاختبار الوحدة في البرمجية

IEEE Std 829-1998, Standard for Software Test Documentation.
معيار لتوثيق اختبار البرمجية

[1] cc2e.com/2050

[2] cc2e.com/2057

نقاط مفتاحية

- الجودة مجانية، في النهاية، لكنها تتطلب إعادة توزيع للموارد بحيث تُمنع العيوب برخص بدلاً من أن تُصلح بغلاء.
- ليست أهداف ضمان الجودة كلها قابلة للإنجاز بنفس الزمن. قرر بشكل صريح أي الأهداف تريد أن تنجز، وأوصل الأهداف إلى الناس الآخرين في فريقك.
- ما من تقنية إزالة عيوب مفردة تكون فعالة بشكل كامل لوحدها. الاختبار لوحده ليس فعال بشكل مثالي في إزالة الأخطاء. تستخدم برامج ضمان الجودة الناجحة عدة تقنيات مختلفة لاكتشاف أنواع مختلفة من الأخطاء.
- تستطيع تطبيق تقنيات فعالة خلال البناء وتقنيات كثيرة قوية بشكل متساوٍ قبل البناء. الإبكار في أن تجد العيوب، يعني، التعقيد الأقل الذي ستكون عليه بقية شفرتك، ويعني، الضرر الأقل الذي ستسببه.
- ضمان الجودة في حلبة البرمجيات هو موجه بالعملية. لا يمتلك تطوير البرمجيات وجه متكرر يؤثر في المنتج النهائي كما يمتلك التصنيع، لذا جودة النتيجة خاضعة لسيطرة العملية المستخدمة في تطوير البرمجية.

الفصل 21

البناء التعاوني

المحتويات[1]

مواضيع ذات صلة

- المنظر الطبيعي الخارجي لجودة البرمجيات: الفصل 20
- اختبار المطور: الفصل 22
- التصحيح: الفصل 23
- متطلبات البناء: الفصل 3 و4

قد تكون خضت تجربة معروفة لكثير من المبرمجين. تمشي إلى غرفة مبرمج آخر وتقول، "هل تمانع بإلقاء نظرة على هذه الشفرة؟ لدي بعض المشاكل فيها." تبدأ بشرح المشكلة: "لا يمكن أن تكون نتيجة لهذا الشيء، لأنني قمت بكذا. ولا يمكن أن تكون نتيجة لهذا الشيء الآخر، لأنني قمت بكذا. ولا يمكن أن تكون نتيجة ل-انتظر دقيقة. يمكن أن تكون نتيجة لذلك. شكراً!" تكون قد حللت مشكلتك قبل أن يمتلك "مساعدك" الفرصة ليقول كلمة.

بطريقة أو بأخرى، كل تقنيات البناء التعاوني هي محاولات لترسيم عملية عرض عملك على شخص آخر لتفريغه من الأخطاء.

إذا كنت قد قرأت عن التفحصات والبرمجة الثنائية من قبل، لن تجد الكثير من المعلومات الجديدة في هذا الفصل. قد تفاجئك البيانات المثبتة عن مدى فعالية التفحصات في القسم 21. 3، وقد لا تكون قد وضعت في

[1] cc2e.com/2185

حسبانك بدائل قراءة الشفرة الموصوفة في القسم 21. 4. قد تلقي نظرة على الجدول 21-1، "مقارنة بين تقنيات البناء التعاوني،" في نهاية هذا الفصل. إذا كانت معرفتك من خبرتك الذاتية، اقرأ! لدى أناس آخرين خبرات مختلفة، وستجد بعض الأفكار الجديدة.

21. 1 نظرة عامة على تطبيقات التطوير التعاوني

يشير "البناء التعاوني" إلى البرمجة الثنائية، والتفحصات الرسمية، والمراجعات التقنية غير الرسمية، وقراءة المستندات، بالإضافة إلى تقنيات أخرى يتشارك المطورون بها المسؤولية لإنشاء شفرة و"منتجات عمل" أخرى. في شركتي، صُكَّ المصطلح "البناء التعاوني" بواسطة مات بيلوكوين في حوالي سنة 2000. ويبدو أن المصطلح قد صُكَّ بشكل مستقل من قبل آخرين في نفس الفترة الزمنية كل على حدة.

بيانات مثبتة

تعتمد كل تقنيات البناء التعاوني، رغم اختلافاتها، على الأفكار: يعمى مطورون عن رؤية بعض من مناطق المشاكل في عملهم، وليس لدى مطورين آخرين نفس مناطق العمى، ومن المفيد للمطورين أن ينظر شخص آخر إلى عملهم. وجدت دراسات في **معهد هندسة البرمجيات** أن المبرمجين حشروا بمتوسط 1 إلى 3 عيوب بالساعة إلى تصاميمهم و5 إلى 8 عيوب بالساعة إلى شفراتهم (هامفري 1997)، لذا مهاجمة هذه المناطق العمياء هي مفتاح للبناء الفعال.

يتمم البناء التعاوني تقنيات ضمان الجودة الأخرى

بيانات مثبتة

الغرض الرئيسي من العمل التعاوني هو تحسين جودة المنتج. كما ذُكر الفصل 20، " المنظر العام لجودة البرمجيات،" *لاختبار البرمجية* فعالية محدودة إذا استخدم لوحده-متوسط معدل اكتشاف الأخطاء حوالي 30 بالمئة فقط لاختبار الوحدة، و35 بالمئة لاختبار التكامل، و35 بالمئة لاختبار بيتا منخفض المستوى. بالمقابل، متوسط فعالية تفحصات التصميم والشفرة هو 55 و60 بالمئة (جونز 1996). الفائدة الثانوية للبناء التعاوني هي أنه يقلل وقت التطوير، الذي بدوره يقلل كُلف التطوير.

نقطة مفتاحية

اقترحت أوائل التقارير في البرمجة الثنائية أنها تستطيع إنجاز جودة شفرة بمستوى مشابه للتفحصات الرسمية (شل وآخرون 2002). ربما كلفة برمجة ثنائية تامّة أعلى من كلفة التطوير الفردي-حوالي 10-25 بالمئة أعلى-لكن يظهر أن الإنقاص في زمن التطوير يكون بحوالي 45 بالمئة، والذي قد يكون في بعض الحالات تقدُّم حاسم على التطوير الفردي (بويم وترنر 2004)، مع أنه ليس فوق التفحصات والتي أعطت نتائج مشابهة.

دُرست المراجعات التقنية لفترة أطول بكثير من البرمجة الثنائية، ونتائجها، كما هي موصوفة في حالات الدراسة وأي مكان آخر، مثيرة للإعجاب:



- وجدت **آي بي إم** أن كل ساعة تفحص منعت حوالي 100 ساعة من عمل ذي صلة (الاختبار وتصحيح الأخطاء) (هولاند 1999).
- خفّضت **رايثيون** كلفة تصحيح العيوب (إعادة العمل) الخاصة بها من حوالي 40 بالمئة من مجمل كلفة المشروع إلى حوالي 20 بالمئة عن طريق مبادرة ركزت على التفحصات (هالي 1996).
- أصدرت **هيلويت-باكارد** تقريراً أن برنامج التفحص الخاص بها وفر 21.5 مليون دولار مقدر بالسنة (غرادي وفان سلاك 1994).
- وجدت **صناعات الكيمياويات الملكية** أن كلفة صيانة مجموعة من حوالي 400 برنامج كانت مرتفعة حوالي 10 بالمئة فقط من مقدار ارتفاع كلفة صيانة مجموعة مشابهة من البرامج لم تكن مُتفحَّصة (غيلب وغراهام 1993)
- وجدت دراسة للبرامج الكبيرة أن كل ساعة تُصرف على التفحصات تُجنِّب ما متوسطه 33 ساعة من أعمال الصيانة، وأن فعالية التفحصات وصلت إلى أكثر ب 20 مرة من فعالية الاختبار (روسيل 1991).
- في منظمة لصيانة البرمجيات، كان 55 بالمئة من تغييرات صيانة "السطر الواحد" خاطئة قبل أن تُقدَّم مراجعات الشفرة. بعد تقديم المراجعات، 2 بالمئة فقط من التغييرات كانت خاطئة (فريدمان ووينبيرغ 1990). عندما اعتُبرت كل التغييرات، 95 بالمئة كانت صحيحة من أول مرة بعد تقديم المراجعات. قبل تقديم المراجعات، أقل من 20 بالمئة كان صحيحاً من المرة الأولى.
- طُورت مجموعة من 11 برنامج بواسطة نفس المجموعة من الناس، وأُصدرت كلها للإنتاج. طُورت أول خمسة بدون مراجعات وامتلكت متوسط 4.5 خطأ بكل مئة سطر من الشفرة. تم تفحص الستة الأخرى فامتلكت متوسط بحوالي 0.82 خطأ بكل مئة سطر من الشفرة. خفّضت المراجعات الأخطاء بأكثر من 80 بالمئة (فريدمان ووينبيرغ 1990).
- أصدر كابيرز جونز تقريراً بأن كل المشاريع البرمجية التي درسها والتي أنجزت معدل 99 بالمئة من إزالة العيوب أو أفضل، كلها استخدمت التفحصات الرسمية. أيضاً، ولم يستخدم أي من المشاريع التي أنجزت فعاليه إزالة عيوب بنسبة أقل من 75 بالمئة التفحصات الرسمية (جونز 2000).

يوضح عدد من هذه الحالات **المبدأ العام لجودة البرمجيات**، والذي يثبّت أن تخفيض عدد العيوب في البرمجية أيضاً يحسن وقت التطوير.



أظهرت دراسات متنوعة أنه بالإضافة إلى كونها أكثر فعالية في القبض على الأخطاء من الاختبار، تجد التطبيقات التعاونية أنواعاً مختلفة عن الأخطاء التي يجدها الاختبار. (مايرز 1978؛ وباسيلي وسيلبي وهاتشن 1986). كما أشار كارل ويدجرز، "يستطيع مراجع بشري أن يُعلِّم رسائل الخطأ الغامضة،

والتعليقات غير الملائمة، وقيم المتحولات "المشفرة بصعوبة"، ونماذج الشفرة المتكررة التي ينبغي أن تكون موحَّدة. لن يقوم الاختبار بذلك" (ويدجيرز 2002). أثر ثانوي هو أن الناس يدققون عملهم بحذر، عندما يعرفون أنه سيراجع. وهكذا، حتى عندما يتم الاختبار بنجاح، تكون المراجعات والأنواع الأخرى من التعاون لازمة كجزء من جودة البرنامج الشاملة.

البناء التعاوني يؤمن القيادة الروحية في الثقافة والخبرة البرمجية التشاركيتين

يمكن أن تدون المعايير البرمجية وتوزع، لكن إن لم يتحدث أحد عنها أو لم يحث الأخرين على استخدامها، فلن تُتبع.[1] المراجعات آلية هامة لإعطاء المبرمجين التغذية الراجعة عن شفرتهم. الشفرة، والمعايير، والأسباب لجعل الشفرة تلاقي المعايير مواضيع جيدة لنقاشات المراجعة.

بالإضافة إلى التغذية الراجعة عن مدى جودة اتباعهم للمعايير، فإن المبرمجين يحتاجون تغذية راجعة عن مفاهيم برمجية معتمدة على أفكارهم أكثر (من الحقائق): التنسيق، والتعليقات، وأسماء المتحولات، واستخدام المتحولات الشاملة والمحلية، ونُهج التصميم، والطريقة التي نفعل بها الأمور حولنا، وهلم جراً. يحتاج المبرمجون المستجدون المبتدؤون إلى إرشاد من هؤلاء الأكثر معرفة، ويحتاج المبرمجون الأكثر معرفة الذين يميلون ليكونوا مشغولين أن يُشجعوا كي يصرفوا وقتاً في مشاركة ما يعرفونه. تخلق المراجعات مكاناً للمبرمجين الأكثر خبرة والمبرمجين الأقل خبرة ليتواصلوا بالمواضيع التقنية. كذلك، فإن المراجعات فرصة لتنمية تطويرات الجودة في المستقبل تماماً كالحاضر.

أصدر أحد الفرق التي استخدمت التفحصات تقريراً أن التفحصات رفعت بسرعة كل المطورين إلى مستوى المطورين الأفضل (تاكيت وفان دورين 1999).

الملكية الجماعية تُطبَّق على كل صيغ البناء التعاوني

بالملكية الجماعية، كل الشفرة تُمتلك من قبل المجموعة بدلاً من أفراد ويمكن أن يتم الدخول إليها وتعديلها من قبل أعضاء المجموعة المختلفين[2]. هذا يقدم عدة منافع قيّمة:

- تشـرق جودة أفضـل للشـفرة من عدة مجموعات من الأعين تنظر إلى الشـفرة وعدة مبرمجين يعملون عليها.

[1] كانت إجرائيات المراجعة غير الرسمية تُمرر من شخص إلى شخص وفق الثقافة العامة للحوسبة لعدة سنين قبل أن يُعترف بذلك بشيء مطبوع. كانت الحاجة للمراجعة واضحة جداً بالنسبة للمبرمجين الأفضل بحيث أنهم نادراً ما ذكروها بالمطبوعات، بينما كان المبرمجون الأسوأ يصدقون أنهم جيدون جداً، فلا يحتاج عملهم إلى مراجعة.

[2] **إشارة مرجعية** المفهوم الذي يشمل كل تقنيات البناء التعاوني هو فكرة الملكية الجماعية. في بعض نماذج التطوير، يمتلك المبرمجون الشفرة التي يكتبونها وتوجد قيود رسمية أو غير رسمية على تعديل شفرة شخص آخر. تزيد الملكية الجماعية الحاجة لتنسيق العمل، خصوصاً إدارة الإعدادات. لتفاصيل، انظر القسم 28. 2، "إدارة الإعدادات"

- يُقلِّل تأثير مغادرة شخص ما للمشروع لأن عدة أشخاص يألفون كل أقسام الشفرة.
- دورات تصحيح العيوب تصبح أقصر بعد كل شيء لأنه يمكن لأي واحد من عدة مبرمجين أن يوظِّف بشكل محتمل لإصلاح الثغرات على أساس "طالما متاح".

بعض المنهجيات، كـ**البرمجة الزائدة**، تنصح بشكل رسمي بتثنية المبرمجين "ترتيب المبرمجين في أزواج" ومناوبة مهام عملهم مع الوقت. في شركتي، وجدنا أن المبرمجين لا يحتاجون أن يُثنَّوا بشكل رسمي لإنجاز تغطية جيدة للشفرة. مع الوقت أنجزنا تغطية متقاطعة من خلال تجميعة من البرمجة الثنائية والمراجعات التقنية الرسمية وغير الرسمية متى لزمت، ومناوبة مهام تصحيح العيوب.

يُطبق التعاون قبل البناء بمقدار تطبيقه بعده

هذا الكتاب عن البناء، لذا التعاون في الشفرة والتصميم المفصلين هما نقطة تركيز هذا الفصل. على أي حال، معظم التعليقات حول البناء التعاوني في هذا الفصل تطبَّق أيضاً على أعمال التخمينات والمخططات والمتطلبات والهيكلة والاختبار والصيانة. بدراسة المراجع في نهاية هذا الفصل، تستطيع أن تطبق التقنيات التعاونية على معظم نشاطات تطوير البرمجية.

2.21 البرمجة الثنائية

في البرمجة الثنائية، أحد المبرمجين يكتب الشفرة بلوحة المفاتيح والأخر يشاهد بحثاً عن الأخطاء ويفكر بطريقة استراتيجية إن كانت الشفرة تُكتب بطريقة صحيحة وإن كانت الشفرة الصحيحة تُكتب. شيعت البرمجة الثنائية في الأصل بواسطة **البرمجة الزائدة** (بيك 2000)، لكنها الآن تستخدم بشكل أوسع (ويليامز وكيسلير 2002).

مفاتيح النجاح بالبرمجة الثنائية

المفهوم الأساسي في البرمجة الثنائية بسيط، لكن مع هذا استخدامه ينتفع من عدة توجيهات:

ادعم البرمجة الثنائية بأعراف كتابة الشفرة لن تكون البرمجة الثنائية فعالة إن كان الشخصين في الثنائية يصرفون وقتهم بالجدال حول أسلوب كتابة الشفرة. حاول أن تضع معايير لما أشار إليه الفصل 5، "التصميم في البناء،" ب "الخصائص العرضية" للبرمجة بحيث يستطيع المبرمجون التركيز على المهمة "الجوهرية" بين أيديهم.

لا تدع البرمجة الثنائية تنقلب إلى مشاهدة ينبغي على الشخص الذي لا يملك لوحة المفاتيح أن يكون مشاركاً فعالاً في البرمجة. ذلك الشخص يحلل الشفرة، ويفكر مباشرة بالذي سيشفر تالياً، ويقيم التصميم، ويخطط كيف يختبر الشفرة.

لا تجبر على البرمجة الثنائية للأشياء السهلة أحد المجموعات التي استخدمت البرمجة الثنائية للشفرة الأكثر تعقيداً وجدت نفعاً أكثر في أن تقوم بتصميم مفصل على اللوح الأبيض ل 15 دقيقة وبعدها تبرمج بطريقة فردية (مانزو 2002). استقرت معظم المنظمات التي تستخدم البرمجة الثنائية على استخدام أزواج في جزء من عملهم وليس كله (بويم وترنر 2004).

ناوب الأزواج ومهام العمل بشكل دوري في البرمجة الثنائية، كما في تطبيقات التطوير التعاوني الأخرى، تظهر المنفعة من تعلُّم مبرمجين مختلفين لأجزاء مختلفة من النظام. ناوب الأزواج بشكل دوري لتحفز التلقيح المتقاطع-بعض الخبراء ينصحون بتغيير الأزواج بمعدل يومي (ريفير 2002).

شجع الأزواج على أن يلاقي كل منهم سرعة تقدم الآخر شريك يتقدم بسرعة كبيرة يحد من فائدة أن يكون له شريك. الشريك السريع مُلزَم بأن يتمهل، وإلا ينبغي أن يُفصل الزوجين وتتم إعادة الترتيب مع أزواج مختلفين.

تأكد أن كلا الشريكين قادران على رؤية الشاشة حتى الأشياء التي تبدو قضايا عادية مثل القدرة على رؤية الشاشة واستخدام خطوط صغيرة جداً يمكن أن تسبب مشاكل.

لا تجبر الناس الذين لا يحبون بعضهم على التشارك أحياناً المشاكل الشخصية تمنع الناس من التشارك بفعالية. إنه أمر فارغ أن تجبر الناس الذين لا يتفقون مع بعضهم على التشارك، لذا كن مرهف الإحساس لتلاقي الشخصيات (بيك 2000، ريفير 2002).

تجنب أزواج كلها مبتدئين تعمل البرمجة اًلثنائية أفضل ما يمكن عندما يكون واحد على الأقل من الشريكين قد شارك من قبل (لارمان 2004).

عين قائد الفريق إذا كان كل فريقك يريد أن يقوم ب 100 بالمئة من البرمجة في أزواج، لا تزال تحتاج أن تعيّن شخصاً واحداً لينسق مهام العمل، ويكون تحت يدك مسؤولاً عن النتائج، ويلعب دور نقطة الاتصال بالنسبة للناس خارج المشروع.

فوائد البرمجة الثنائية

تؤدي البرمجة الثنائية إلى عدة فوائد:

- صمودها تحت الضغط أفضل من البرمجة الفردية. الزوجين يشجعون بعضهم على المحافظة على جودة الشفرة عالية حتى عندما يوجد ضغط لكتابة شفرة سريعة ومتسخة.
- إنها تحسن جودة الشفرة. تميل قابلية قراءة وفهم الشفرة إلى الارتفاع إلى مستوى المبرمج الأفضل في الفريق.
- إنها تقصّر جداول المواعيد. يميل الزوجين لكتابة شفرة بسرعة أكبر وبأخطاء أقل. يصرف فريق المشروع وقتاً أقل في نهاية المشروع على تصحيح الأخطاء.
- إنها تؤدي إلى كل الفوائد العامة الأخرى للبناء التعاوني، متضمنة: نشر الثقافة التشاركية، والأخذ بيد المبرمجين الحديثين، وتبنّي الملكية الجماعية.

لائحة اختبار: البرمجة الثنائية الفعالة[1]

- هل لديك عرف كتابة شفرة بحيث يبقى زوج المبرمجَين مركزين على البرمجة بدلاً من الجدالات الفلسفية حول أسلوب كتابة الشفرة.
- هل يتشارك كلا الشريكين بشكل فعال؟
- هل تتجنب البرمجة الثنائية لكل شيء و، بدلاً، تختار المهام التي حقاً ستستفيد من البرمجة الثنائية؟
- هل تُناوب الأزواج ومهام العمل بشكل دوري؟
- هل يتلاقى الزوجين في سرعة التقدم والأمور الشخصية؟
- هل يوجد قائد للفريق ليكون النقطة المركزية بالنسبة للإدارة والناس خارج المشروع؟

[1] cc2e.com/2192

3.21 التفحصات الرسمية

التفحص هو نوع محدد من المراجعات والتي وُجد أنها فعالة جداً في اكتشاف العيوب ونسبياً اقتصادية بالمقارنة إلى الاختبار. طُوِّرت التفحصات من قبل مايكل فاجان واستُخدمت في IBM لعدة سنين قبل أن ينشر فاجان المقالة التي جعلتها عامة. رغم أن أي مراجعة تتضمن قراءة للتصاميم أو الشفرة، يختلف التفحص عن المراجعة العادية بعدة طرق مفتاحية:

- تركز لوائح الاختبار انتباه المراجعين على مناطق كانت تحوي مشاكل في الماضي.
- يركز التفحص على اكتشاف الأخطاء، لا تصحيحها.
- يحضّر المراجعون لاجتماع التفحص مسبقاً ويصلون مع لائحة من المشاكل التي اكتشفوها.
- تُسنَد أدوار متباينة لكل المشتركين.
- ليس رئيس التفحص هو مالك منتج العمل (البرمجية) تحت التفحص.
- قد تلقّى الرئيس تدريباً خاصاً في رئاسة التفحصات.
- يُقام اجتماع التفحص فقط إذا كان كل المشتركين جاهزين بشكل كاف.
- تُجمع البيانات في كل تفحص وتزوَّد إلى التفحصات المستقبلية لتحسينها.
- لا تحضـر الإدارة العامة اجتماع التفحص مالم تكن تتفحص مخطط مشـروع أو مواد إدارية أخرى. بل يحضر القادة التقنيين.

ما النتائج التي تستطيع أن تتوقها من التفحصات؟

تجد التفحصات الفردية عادةً حوالي 60 بالمئة من العيوب، والتي هي أعلى من التقنيات الأخرى ما عدا إعطاء النماذج الأولية واختبار بيتا كبير الحجم. أكدت هذه النتائج مرات عديدة في منظمات مختلفة، متضمنة **هارس بي سي اس دي، وتجربة جودة البرمجيات الوطنية، ومعهد هندسة البرمجيات، وهيلويت باكارد،** إلخ (شل وآخرون 2002).

تزيل تجميعة تفحصات للتصميم والشفرة عادةً 70-85 بالمئة أو أكثر من العيوب في المنتج (جونز 1996).

تحدد التفحصات الصفوف الميّالة للأخطاء في وقت مبكر، وقد كتب كابرز جونز أنها أنتجت عيوب أقل ب 20-30 بالمئة لكل 1000 سطر من الشفرة بالمقارنة مع تطبيقات مراجعة أقل رسمية. يتعلم المصممون والمشفرون أن يحسنوا عملهم من خلال الاشتراك في التفحصات، وتزيد التفحصات الإنتاجية بحوالي 20 بالمئة (فاجان 1976، وهامفري 1989، وغيلب وغراهام 1993، وويجرز 2002). في مشروع يستخدم التفحصات

للتصميم والشفرة، ستأخذ التفحصات حوالي 10-15 بالمئة من ميزانية المشروع وستخفض بالحالة الطبيعية كلفة المشروع الكلية.

يمكن أن تُستخدم التفحصات أيضاً لتخمين التقدم، لكن الذي سيُخمَّن هو التقدم التقني. وهذا عادةً يعني الإجابة على سؤالين: هل عُمل العمل التقني؟ وهل العمل التقني يُعمل بشكل جيد؟ الإجابتين على السؤالين هي ناتج ثانوي للتفحصات الرسمية.

الأدوار خلال التفحص

إحدى المميزات المفتاحية للتفحص هي ان كل شخص ضمن التفحص يلعب دوراً فريداً. إليك هذه الأدوار:

الرئيس الرئيس هو المسؤول عن الحفاظ على تقدم التفحص بمعدل سريع كفاية حتى يكون مثمر وبطيء كفاية حتى يتم إيجاد معظم الأخطاء المحتملة. يتوجب على الرئيس أن يكون كفؤا من الناحية التقنية-ليس خبيراً في التصميم أو الشفرة المحددين تحت التفحص بالضرورة، لكن قادراً على فهم التفاصيل ذات الصلة. يدير هذا الشخص الجوانب الأخرى للتفحص، مثل توزيع التصميم أو الشفرة للمراجعة، وتوزيع لائحة اختبار التفحص، وتهيئة غرفة الاجتماع، وكتابة تقارير عن نتائج التفحص، ومتابعة بنود المبادرات المُسنَدة في اجتماع التفحص.

الكاتب يلعب الشخص الذي كتب التصميم أو الشفرة دوراً غير هام نسبياً في التفحص. أحد جوانب الهدف من التفحص أن تتأكد من أن التصميم أو الشفرة تتكلم بمفردها. إذا خَلُص إلى كون التصميم أو الشفرة تحت التفحص غامضاً، سيسنَد إلى الكاتب وظيفة جعله أوضح. وإلا، فواجبات الكاتب أن يشرح أقسام غامضة من التصميم أو الشفرة و، أحياناً، يشرح لمَ الأشياء التي تبدو كأنها أخطاء هي بالفعل مقبولة. إذا كان المشروع غير مألوف للمراجعين، قد يقدم الكاتب نظرة عامة عن المشروع في اجتماع التحضير للتفحص.

المراجع المراجع هو أي شخص لديه اهتمام مباشر بالتصميم أو الشفرة لكنه ليس الكاتب. قد يكون مراجع التصميم هو المبرمج الذي سيطبق التصميم. قد يُشرَك أيضاً مختبر أو مُهيكِل مستوى أعلى. دور المراجعين هو اكتشاف العيوب. عادةً يجدون عيوباً خلال التحضير، و، بمناقشة التصميم أو الشفرة في اجتماع التفحص، ينبغي أن تجد المجموعة عيوب أكثر بشكل معتبر.

المُدوِّن يسجل المدون الأخطاء المكتشفة ومهام بنود المبادرة خلال اجتماع التفحص. لا ينبغي للكاتب ولا للرئيس أن يكون مدوناً.

الإدارة عادةً تضمين الإدارة في التفحصات ليس فكرة جيدة. النقطة في التفحص البرمجي هي أن تكون مراجعة تقنية بنقاء. حضور الإدارة يغير التفاعلات: يشعر الناس أنهم، بدلاً من مواد المراجعة، تحت التقييم، والذي يحول التركيز من تقني إلى سياسي. على أي حال، تمتلك الإدارة الحق في أن تعرف نتائج التفحص، ويُحضَّر تقرير التفحص ليبقي الإدارة على علم.

بشكل مشابه، لا ينبغي تحت أي ظرف أن تُستخدم نتائج التفحص لتقييم الأداء. لا تقتل الإوزة التي تضع بيضاً ذهبياً. لا تزال الشفرة الممتحنة بالتفحص تحت التطوير. تقييم الأداء ينبغي أن يعتمد على المنتجات النهائية، وليس على عمل لم ينتهِ.

بعد كل ذلك، ينبغي ألا يتضمن التفحص أقل من ثلاثة مشتركين. لا يمكن أن تجد رئيساً وكاتباً ومراجعاً متباينين بأقل من ثلاثة أشخاص، ولا ينبغي لهذه الأدوار أن تُجمع. نصيحة قديمة: أن تحدد التفحص بحوالي ستة أشخاص لأنه، مع أية زيادة، تصبح المجموعة كبيرة جداً على أن تُدار. وجد باحثون أن، عموماً، وجود أكثر من اثنين أو ثلاثة مراجعين لا يبدو أنه يزيد عدد الأخطاء المكتشفة (بوش وكيلي 1989، وبورتر وفوتّا 1997). على أي حال، ما من إجماع على هذه النتائج العامة، وتبدو النتائج أنها تتغير حسب نوع المادة تحت التفحص (ويدجرز 2002). انتبه إلى خبرتك، وعيّر نهجك وفقها.

الإجرائية العامة للتفحص

يتألف التفحص من عدة مراحل متباينة:

التخطيط يعطي الكاتب التصميم أو الشفرة إلى الرئيس. يقرر الرئيس من سيراجع المادة ومتى وأين سيعقد اجتماع التفحص، يوزع الرئيس بعد ذلك التصميم أو الشفرة، ولائحة مهام تركز تنبيه المتفحصين. ينبغي للمواد أن تُطبع مع أرقام الأسطر لتسريع تحديد الخطأ خلال الاجتماع.

النظرة العامة عندما لا يكون المراجعون آلفين للمشروع الذي يراجعونه، يمكن أن يصرف الكاتب وقتاً أعلاه ساعة أو بحدودها، ليصف البيئة التقنية التي أُنشئ فيها التصميم أو الشفرة. يميل إجراء نظرة عامة لأن يكون تطبيقاً خطيراً لأنه يمكن أن يؤدي إلى ستر النقاط الغامضة في التصميم أو الشفرة تحت التفحص. ينبغي أن يتكلم التصميم أو الشفرة لوحدهما؛ لا ينبغي أن تتكلم النظرة العامة عنهما.

التحضير [1] يعمل كل مراجع بمفرده ليمتحن التصميم أو الشفرة بحثاً عن الأخطاء. يستخدم المراجعون لائحة المهام ليتبعوا ويوجّهوا امتحانهم لمواد المراجعة.

لمراجعة شفرة تطبيق مكتوبة بلغة عالية المستوى، يستطيع المراجعون أن يحضّروا حوالي 500 سطر من الشفرة في الساعة. لمراجعة شفرة نظام مكتوبة بلغة عالية المستوى، يستطيع المراجعون التحضير بحوالي 125 سطر من الشفرة في الساعة فقط (هامفري 1989). يغيّر معدلُ الفعالية العظمى للمراجعة الصفقة الرابحة، لذا احتفظ بسجلات معدلات التحضير في منظمتك لتحدد المعدل الأكثر فعالية في بيئتك.

وجدت بعض المنظمات أن التفحصات تكون أكثر فعالية عندما يُسند لكل مراجع منظور محدد. قد يُسأل المراجع أن يحضّر التفحص من وجهة نظر مبرمج الصيانة، أو الزبون، أو المصمم، على سبيل المثال. لم تكن الأبحاث في المراجعات المعتمدة على المنظور شاملة، لكنها توحي أن المراجعات المعتمدة على المنظور قد تكتشف أخطاء أكثر من المراجعات العامة.

تنوع إضافي في التحضير للتفحص هو أن تُسند سيناريو أو أكثر لكل مراجع كي يفحصه. يمكن أن يتضمن السيناريو أسئلة محددة على المراجع الإجابة عليها، مثل "هل تتواجد أية متطلبات غير مستوفاة في هذا التصميم؟" أيضاً قد يتضمن السيناريو مهام محدد على المراجع أن ينجزها، مثل تعداد المتطلبات المحددة التي استوفتها مواد التصميم المعين. تستطيع أيضاً أن تسند قراءة المادة من الوراء إلى الامام أو من الداخل إلى الخارج لبعض المراجعين.

اجتماع التفحص يختار الرئيس شخصاً غير الكاتب ليعبّر بكلمات سهلة عن التصميم أو يقرأ الشفرة (ويدجرز 2003). يُشرح كل المنطق، بما في ذلك كل فرع من كل تركيب منطقي. خلال هذا التقديم، كلما اكتشفَت أخطاء يسجّلها المدون، لكن النقاش حول الخطأ يتوقف بمجرد التعرف عليه على أنه خطأ. يضع المدون ملاحظات حول نوع وخطورة الخطأ، ويتقدم التفحص. إذا واجهتَ مشاكل في الحفاظ على النقاش مركزاً، قد يرن الرئيس جرس ليلفت انتباه المجموعة ويعيد النقاش إلى طريقه.

ينبغي ألا يكون معدل التفكير بالشفرة أو بالتصميم بطيئاً جداً ولا سريعاً جداً. إذا كان بطيئاً جداً، يمكن أن يفتر الانتباه ولن يكون الاجتماع منتجاً. إذا كان سريعاً جداً، يمكن أن تغفل المجموعة عن أخطاء كانت ستكتشف لو لم يكن كذلك. تتغير معدلات التفحص المثالية من بيئة إلى بيئة، تماماً كما تفعل معدلات التحضير. احتفظ بالسجلات حتى تستطيع مع مرور الزمن أن تحدد المعدل المثالي لبيئتك. وجدت منظمات أخرى أنه من أجل شفرة نظام، معدل تفحص ب 90 سطر من الشفرة في الساعة هو معدل مثالي. من أجل شفرة التطبيقات، يمكن أن يكون معدل التفحص بسرعة 500 سطر من الشفرة في الساعة (هامفري 1989). متوسط بحوالي 150-200 عبارة مصدرية، ليست تعليقاً وليست فارغة، بالساعة هو منطلق جيد للبداية (ويدجرز 2002).

[1] **إشارة مرجعية** من أجل لائحة مهام تستطيع استخدامها لتحسين جودة الشفرة انظر الصفحة xxix.

لا تناقش الحلول خلال الاجتماع. ينبغي أن تبقى المجموعة مركزة على تمييز الأخطاء. بعض مجموعات التفحص لا تسمح حتى بالنقاش إن كان عيب هو عيب فعلاً. يعتبرون أنه إن كان شخص ما مرتبكاً كفاية ليفكر أنه عيب، فإن التصميم أو الشفرة أو التوثيق يحتاج إلى أن يوضَّح.

ينبغي ألا يستمر الاجتماع أكثر من ساعتين. هذا لا يعني أن تلفّق إنذار حريق لتخرج الجميع عند إشارة الساعتين، لكن الخبرة في **أي بي إم** وشركات أخرى كانت أن المراجعين لا يستطيعون التركيز لأكثر من حوالي ساعتين في كل مرة. لنفس السبب، فإنه من غير الحكمة أن تنظم أكثر من تفحص في نفس اليوم.

تقرير التفحص في اليوم الذي انعقد فيه اجتماع التفحص، يقدم رئيس التفحص تقريراً (بريد إلكتروني أو شيء معادل) يعدد فيه كل الأخطاء، متضمناً نوع ودرجة خطورة كل خطأ. يساعد تقرير التفحص على التأكد من أن كل العيوب ستُصحَّح، ويستخدم لتطوير لائحة اختبار تشدد على المشاكل الخاصة بالمنظمة. إذا جمعت بيانات عن الوقت المصروف وعدد الأخطاء المكتشفة خلال الوقت، تستطيع أن تستجيب للتحديات المتعلقة بفعالية التفحص ببيانات مثبته. وإلا، ستكون مقيداً بان تقول أن التفحصات تبدو أفضل. هذا لن يكون مقنعاً لشخص يعتقد أن الاختبار يبدو أفضل *(الإشارة المضمنة هنا إلى حاجة الثاني إلى اثبات فهو بالأساس يظن ذلك).* ستستطيع أن تخبر إن كانت التفحصات لا تعمل في بيئتك فتغيرها أو تتنازل عنها، كما هو مناسب. جمع البيانات مهم أيضاً لأن أي منهجية جديدة تحتاج أن تبرر وجودها.

إعادة العمل يسند الرئيس العيوب لشخص ما، عادة الكاتب، كي يصلحها. يصلح المعهود إليه كل عيب في اللائحة.

المتابعة الرئيس مسؤول عن رؤية كل العمل المعاد الذي أسند خلال التفحص قد أُنجز. بالاعتماد على عدد الأخطاء المكتشفة وخطورة هذه الأخطاء، قد يتابع بجعل المراجعين يعيدون التفحص لكامل منتج العمل، أو بجعل المراجعين يعيدون التفحص للإصلاحات فقط، أو بالسماح للكاتب بأن يكمل إصلاحاته بدون أي عمل لاحق.

اجتماع الثلث ساعة على الرغم من أنه خلال الاجتماع لا يُسمح للمشاركين في التفحص أن يناقشوا حلول المشاكل البارزة، فقد لا يزال البعض يريد ذلك. تستطيع أن ترتب اجتماعاً غير رسمي، لثلث ساعة كي تسمح بمناقشة الحلول للمجموعات المهتمة بعد نهاية التفحص الرسمي.

المعايرة النهائية للتفحص

حالما تصبح ماهراً في أداء التفحصات "بفضل الكتاب،" تستطيع عادة أن تجد عدة طرق لتحسن هذه التفحصات. لا تجبر نفسك على القيام بتغييرات، مع ذلك. "أدِر" عملية التفحص بحيث تعرف إن كانت تغييراتك مفيدة.

وجدت الشركات أن إزالة أو جمع أي من المراحل غالباً يكلف أكثر مما يوفره (فاجان 1986). إذا كنت مفتون بتغيير عملية التفحص بدون قياس أثر التغيير، فلا تفعل. إذا قِستَ العملية وعرفت أن عملية التغيير تعمل بشكل أفضل من العملية المشروحة هنا، تقدَّم مباشرة.

وأنت تقوم بالتفحصات، ستلاحظ أن أنواع معينة من الأخطاء تحدث بتكرار أكثر من الأنواع الأخرى. أنشئ لائحة اختبار تستدعي الانتباه إلى تلك الأنواع من الأخطاء بحيث يركز المراجعون عليها. مع مرور الوقت، ستجد أنواع من الأخطاء غير موجودة في لائحة الاختبار، أضفها. قد تجد أن بعض الأخطاء في اللائحة الابتدائية توقفت عن الحدوث، إزلها. بعد عدة تفحصات، سيكون لدى منظمتك لائحة اختبار للتفحصات المعدَّلة وفق حاجاتها، وقد تمتلك دليل عن مناطق الإشكال التي يحتاج المبرمجون تدريباً أو دعماً أكثر فيها. حدد لائحة اختبارك بصفحة واحدة أو أقل. الأطول صعبة الاستخدام بمستوى التفصيل اللازم في التفحص.

الأنا في التفحصات

الغاية من التفحص نفسه أن تكتشف العيوب في التصميم أو الشفرة. وليست أن تكتشف البدائل أو تجادل حول من على حق ومن على باطل[1]. الغاية بالتأكيد ليست أن تنتقد كاتب التصميم أو الشفرة. ينبغي أن تكون التجربة إيجابية للكاتب والتي يكون من الواضح فيها أن الاشتراك في المجموعة يحسن البرنامج ويعطي خبرة لكل المنتسبين. لا ينبغي أن تقنع الكاتب أن بعض الناس في المجموعة حمقى أو إنه الوقت للبحث عن عمل جديد. تعليقات مثل "أي شخص يعرف جافا يعرف أنه ذو فعالية أكثر أن تسير الحلقة من 0 إلى *num-1*، وليس من 1 إلى *num*" هي كلياً غير ملائمة، وإن حدثت، ينبغي على الرئيس أن يجعل عدم ملاءمتها واضح بشكل لا يمكن أن يُفهم خطأ.

لأن التصميم أو الشفرة أحدهما يُنتقد وقد يشعر الكاتب أنه بطريقة ما مرتبط به، بشكل طبيعي سيشعر الكاتب ببعض الحرارة الموجهة إليه من الشفرة. ينبغي أن يتوقع الكاتب سماع انتقادات لعدة عيوب ليست بالحقيقة عيوب وعدة أخرى تبدو نقاط خلافية. بغض النظر عن ذلك، ينبغي أن يعترف الكاتب بكل عيب مزعوم ويتقدم.

[1] **اقرأ أيضاً** لنقاش حول البرمجة البعيدة عن الذات، انظر *The Psychology of Computer Programming, 2d ed*. (وينبيرغ 1998).

لا يعني الاعتراف بالنقد أن الكاتب يوافق محتوى النقد. لا ينبغي على الكاتب أن يحاول أن يدافع عن العمل تحت المراجعة. بعد المراجعة، يستطيع الكاتب أن يفكر لوحده بكل نقطة ويقرر إن كانت صالحة.

يتوجب على المراجعين أن يتذكروا أن الكاتب يمتلك المسؤولية المطلقة عن تقرير ما يُفعل بشأن العيب. إنه من الجيد أن تستمتع بإيجاد العيوب (وخارج المراجعة، أن تستمتع باقتراح الحلول)، لكل يتوجب على كل مراجع أن يحترم حق الكاتب المطلق بتقرير كيفية حل خطأ.

التفحصات والشفرة الكاملة

لدي خبرة شخصية باستخدام التفحصات في النسخة الثانية من *الشفرة الكاملة*. في النسخة الأولى من هذا الكتاب، كتبت في البداية مسودة خشنة. بعد الإبقاء على مسودة كل فصل قابعة في الجارور لأسبوع أو أكثر، أعدت قراءة الفصل بلا مشاعر وصححت الأخطاء التي وجدتها. ثم مرّرت الفصل المنقح إلى حوالي دزينة قرناء للمراجعة، راجعه العديد منهم بتعمق تام. صححت الأخطاء التي وجدوها. بعد عدة أسابيع أُخَر، أعدت مراجعته مجدداً لوحدي وصححت أخطاء أكثر. أخيراً، أرسلت المخطوطة إلى الناشر، حيث تمت مراجعتها بواسطة محرر النسخة ومحرر تقني و "القارئ الممتحن" (مصحح الأخطاء). وكانت نسخة مطبوعة من الكتاب من أكثر من 10 سنوات، وأرسل القراء حوالي 200 تصحيح خلال ذلك الزمن.

قد تعتقد أنه لن تبقى أخطاء كثيرة في الكتاب لأنها ذهبت خلال كل نشاط المراجعة ذلك. لكن لم تكن هذه هي الحالة. كي أنشئ النسخة الثانية، استخدمت التفحصات الرسمية للنسخة الأولى لتحديد المشاكل التي تحتاج لحل في النسخة الثانية. جُهّز فرق من ثلاثة إلى أربعة مراجعين حسب التوجيهات الموصوفة في هذا الفصل. مما أثار دهشتي نوعاً ما، أن تفحصاتنا الرسمية وجدت عدة مئات من الأخطاء في نص النسخة الأولى لم تُكتشف مسبقاً من خلال أي من نشاطات المراجعة المتعددة.

إن كان لدي أي شكوك حول قيمة التفحصات الرسمية، تجربتي في إنشاء النسخة الثانية من *الشفرة الكاملة* أزالتها.

ملخص التفحص

تشجع لوائح اختبار التفحص على التركيز المكثف. عملية التفحص هي عملية خاضعة للنظام بسبب لوائحها المعيارية للاختبار وقواعدها المعيارية. إنها أيضاً تتحسن ذاتياً لأنها تستخدم حلقة تغذية راجعة رسمية لتطور

لوائح الاختبار وتراقب معدلات التحضير والتفحص. مع هذا التحكم بالعملية والتحسين المستمر، تصبح التفحصات بسرعة تقنية قوية بغض النظر عن كيف بدأت.

عرّف **معهد هندسة البرمجيات نموذج نضوج المقدرة**. [1]والذي يقيس فعالية عملية تطوير البرمجية في المنظمة (**معهد هندسة البرمجيات** 1995). تشرح عملية التفحص كيف يبدو المستوى الأعلى. العملية خاضعة لنظام وقابلة للتكرار وتستخدم تغذية راجعة مُقاسة لتحسن نفسها. تستطيع أن تطبق نفس الأفكار على العديد من التقنيات الموصوفة في هذا الكتاب. عندما تُعمَّم إلى كادر التطوير بكامله، تكون هذه الأفكار، بإيجاز، هي ما يتولى نقل الكادر إلى أعلى مستوى ممكن من الجودة والإنتاجية.

لائحة اختبار: التفحصات الفعالة2

- هل لوائح الاختبار لديك تركز انتباه المراجعين على مناطق كانت فيها مشاكل في الماضي؟
- هل ركزّتَ التفحص على اكتشاف الأخطاء بدلا من تصحيحها؟
- هل وضعت في حسبانك تعيين وجهات نظر أو سيناريوات لتساعد المراجعين على التركيز على عملهم التحضيري؟
- هل أعطي المراجعون وقتاً كافياً للتحضير قبل اجتماع التفحص، وهل قام كل منهم بالتحضير؟
- هل لدى كل مشترك دور فريد ليلعبه-الرئيس أو مراجع أو مدوّن أو ما إلى ذلك؟
- هل يسير الاجتماع بمعدل مثمر؟
- هل حُدد الاجتماع بساعتين؟
- هل تلقى كل المشاركين تدريب مخصص للتصرف في التفحصات، وهل تلقى الرئيس تدريب خاص في مهارات الرئاسة؟
- هل جُمعت بيانات عن أنواع الأخطاء في كل تفحص بحيث تستطيع حياكة لوائح الاختبار المستقبلية في منظمتك؟
- هل جُمعت بيانات عن معدلات التفحص والتحضير بحيث تستطيع أن تحسن التفحصات والتحضيرات المستقبلية؟
- هل تمت متابعة بنود العمل المعينة في كل تفحص، إما شخصياً من قبل الرئيس أو بإعادة تفحص؟
- هل تتفهم الإدارة أنه لا ينبغي أن تحضر اجتماع التفحص؟
- هل يوجد مخطط متابعة لضمان أن الإصلاحات تمت بشكل صحيح؟

[1] اقرأ أيضاً لتفاصيل أكثر عن مفهوم معهد هندسة البرمجيات المتعلق بالنضج التطوري، انظر Managing the Software Process (هامفري 1989)

[2] cc2e.com/2199

4.21 أنواع تطبيقات التطوير التعاوني الأخرى

الأنواع الأخرى من التعاون لم تزيد هيكل الدعامة التجريبية التي زادت فيها التفحصات والبرمجة الثنائية، لذا غُطّيت هنا بعمق أقل. تتضمن التعاونيات التي غُطيت في هذا القسم: العبورات وقراءة الشفرة و "عروض الكلب والفرس الصغير".[1]

العبورات

العبور هو نوع شائع من المراجعات. المصطلح مُعرَّف بفضاضة، وعلى الأقل بعض من شعبيته يمكن أن يعزا إلى حقيقة أن الناس يستطيعون أن يسموا فعلياً إي نوع من المراجعات "عبوراً".

لأن المصطلح معرَّف بفضاضة تامة، من الصعب القول بالضبط ما هو العبور. بالتأكيد العبور يتضمن اثنين أو أكثر من الناس يناقشون التصميم أو الشفرة. قد يكون برسمية جلسة ثور ارتجالية حول اللوح الأبيض؛ قد يكون برسمية اجتماع مُجدول مع عرض عام مُحضَّر من قسم الفن وملخص رسمي يُرسل إلى الإدارة. بأحد المعاني، "أينما وجد اثنان أو ثلاثة مجتمعون مع بعضهم." يوجد عبور. يحب مؤيّدو العبورات فضاضة هكذا تعريف، لذا سأذكر عدة أشياء تتشارك بها كل العبورات وأترك بقية التفاصيل لك:

- عادة يستضاف ويدار العبور من قبل كاتب التصميم أو الشفرة تحت المراجعة
- يركز العبور على القضايا التقنية-إنه اجتماع عمل.
- يتجهز كل المشاركون للعبور بقراءة التصميم أو الشفرة والبحث عن الأخطاء.
- العبور هو فرصــة للمبرمجين ذوي الدرجة العالية ليمرروا خبرتهم وثقافتهم التشـاركية إلى المبرمجين الجدد. وهو أيضاً فرصة للمبرمجين الجدد أن يقدموا منهجيات جديدة ويتحدَّوا الافتراضات القديمة وربما منتهية الصلاحية.
- يستمر العبور عادة من 30 إلى 60 دقيقة.
- التأكيد على اكتشاف الأخطاء وليس تصحيحها.
- لا تحضر الإدارة.
- مفهوم العبور مرن ويمكن أن يتكيف إلى الحاجات المخصصة للمنظمة التي تستخدمه.

ما النتائج التي يمكن أن تتوقعها من العبور؟

[1] **مصطلحات** عروض الكلب والفرس الصغير (dog and pony shows) مصطلح يدل على الأحداث التي تقام فقط للتأثير بالناس كي يشتروا أو يقدموا الدعم.

عندما يُستخدم بذكاء وانضباط، يمكن للعبور أن يؤدي إلى نتائج مشابهة لنتائج التفحص-هذا يعني، يمكن عادةً أن يجد بين 20 و40 بالمئة من الأخطاء في البرنامج (مايرز 1979، بويم 1987b، يوردون 1989b، جونز 1996). لكن بالعموم، وُجد أن العبورات أقل فاعلية بشكل ملحوظ من التفحصات (جونز 1996).

عندما يُستخدم بطريقة غير ذكية، سيزعجك العبور أكثر مما يفيدك. النهاية الدنيا للفعالية، 20 بالمئة، لا تغني كثيراً، ووجدت منظمة واحدة على الأقل (**خدمات الحواسيب في بوينغ**) أن مراجعات القرناء للشفرة "باهظة الثمن." وجدت **بوينغ** أنه من الصعب أن تحث طاقم الموظفين على التطبيق المتماسك لتقنيات العبور، وعندما يزداد ضغط المشروع، تصبح العبورات تقريباً مستحيلة (غلاس 1982).

أصبحث أكثر انتقاداً للعبورات خلال السنوات العشر الماضية كنتيجة لما رأيت في الأعمال الاستشارية في شركتي. وجدت أنه عندما يكون أناس قد امتلكوا خبرات في المراجعات التقنية، فإنها تقريباً دائماً في التطبيقات غير الرسمية كالعبورات بدلاً من التفحصات الرسمية. المراجعة هي بشكل أساسي اجتماع والاجتماعات مكلفة. إذا كنت ستقوم بتحمّل الأعباء الاعتيادية لعقد اجتماع، فإنه جدير بالاهتمام أن تهيكل الاجتماع بشكل مشابه للتفحص الرسمي. إن كان العمل المُقدَّم الذي تقوم بمراجعته لا يسوغ أعباء تفحص رمسي، فإنه لا يسوغ أعباء الاجتماع على الإطلاق. في هكذا حالة ستكون بوضع أفضل عندما تستخدم قراءة المستند أو أي نهج أقل تفاعلاً.

تبدو التفحصات أكثر فعالية في إزالة الأخطاء من العبورات. لذا لمَ على أي كان أن يختار استخدام العبورات؟

إذا كان لديك مجموعة مراجعة كبيرة، العبور هو خيار مراجعة جيد لأنه يجلب العديد من وجهات النظر المتعارضة لتتصارع على المادة تحت المراجعة. إذا استطاع كل شخص ضمن العبور أن يقتنع أن الحل على ما يرام، فإنه من المحتمل أنه لا يحوي أي عيوب رئيسية.

إذا ضُمّ مراجعون من منظمات أخرى، قد يكون العبور أيضاً مفضلاً. الأدوار في التفحص أكثر رسمية وتتطلب بعض التمرين قبل أن يؤديها الناس بفعالية. المراجعون الذين لم يشتركوا في تفحصات مسبقاً هم عائق. إذا كنت تريد أن تلتمس تبرعاتهم، قد يكون العبور الخيار الأفضل.



التفحصات أكثر تركيزاً من العبورات وعموماً عائدها أفضل. إذن، إذا كنت تختار معيار مراجعة لمنظمتك، اختر التفحصات في الأول مالم يكن لديك سبب جيد لألّا تفعل.

قراءة الشفرة

قراءة الشفرة هي بديل للتفحصات والعبورات. في قراءة الشفرة، تقرأ الشفرة المصدرية باحثاً عن الأخطاء. وتترك تعليقات على الجوانب النوعية من الشفرة، مثل تصميمها وأسلوبها وقابلية قراءتها وقابلية صيانتها وفعاليتها.

بيانات مثبتة

وجدت دراسة في **مختبر ناسا لهندسة البرمجيات** أن قراءة الشفرة اكتشفت حوالي 3.3 عيب في ساعة من الجهد. اكتشف الاختبار حوالي 1.8 خطأ في الساعة (كارد 1987). أيضاً وجدت قراءة الشفرة 20 إلى 60 بالمئة أخطاء أكثر خلال حياة المشروع بالمقارنة مع ما وجدت الأنواع المختلفة من الاختبار.

مثل فكرة العبور، مفهوم قراءة الشفرة معرف بفضاضة. تتضمن قراءة الشفرة عادةً شخصين أو أكثر يقرؤون الشفرة بشكل منفصل ثم يجتمعون مع كاتب الشفرة لنقاشها. إليك كيف تسير أمور قراءة الشفرة:

- في التحضير للاجتماع، يوزع الكاتب مجاناً اللوائح المصدرية على قارئي الشفرة. اللوائح تكون من 1000 إلى 10000 سطر من الشفرة؛4000 سطر هو القياس.
- يقرأ شخصين أو أكثر الشفرة. استعمل شخصين على الأقل كي تشجع المنافسة بين المراجعين. إذا استعملت أكثر من اثنين، قس إسهام كل واحد لتعرف كم يسهم الناس المضافون.
- يقرأ المراجعون الشفرة بشكل منفصل. خمّن معدل حوالي 1000 سطر في اليوم.
- عندما ينتهي المراجعون من قراءة الشفرة، يستضيف كاتب الشفرة اجتماع قراءة الشفرة. يستمر الاجتماع لساعة أو اثنتين ويركز على المشاكل المكتشفة من قبل قارئي الشفرة. ما من أحد يقوم بمحاولة عبور الشفرة سطراً سطراً. حتى إن الاجتماع ليس ضرورياً بصرامة.
- يصلح كاتب الشفرة المشاكل المكتشفة من قبل المراجعين.

نقطة مفتاحية

الفرق بين قراءة الشفرة من جهة والتفحصات والعبورات من جهة أخرى هو أن قراءة الشفرة تركز على المراجعة الفردية للشفرة أكثر من الاجتماع. النتيجة هي أن وقت كل مراجع يتمحور على اكتشاف المشاكل في الشفرة. يُصرف وقت أقل في الاجتماعات والتي فيها يساهم كل فرد بجزء فقط من الوقت والتي فيها يذهب مقدار كبير من الجهد في ديناميكا مجموعة الإشراف. يُصرف وقت أقل في تأخير الاجتماعات حتى يستطيع كل شخص أن يجتمع لساعتين. قراءات الشفرة قيّمة بشكل خاص في حالات تبعثر المراجعين جغرافياً.

بيانات مثبتة

وجدت دراسة ل 13 مراجع في **إيه تي و تي** أن أهمية اجتماع المراجعة بذاته كانت مبالغ بها؛ 90 بالمئة من العيوب اكتُشفت في التحضير لاجتماع المراجعة، وفقط حوالي 10 بالمئة اكتُشفت خلال المراجعة نفسها (فوتا 1991، وغلاس 1999).

عروض الكلب والفرس الصغير

عروض الكلب والفرس الصغير هي مراجعات يُشرح فيها منتج البرمجية لزبون. مراجعة الزبون شائعة في البرمجيات المطورة لصالح التعهدات الحكومية، والتي يشترط فيها أن تقام مراجعات للمتطلبات والتصميم والشفرة. الغرض من عرض الكلب والفرس الصغير أن تشرح للزبون أن المشروع "تمام"، لذا فهو مراجعة إدارية أكثر من كونه مراجعة تقنية.

لا تعتمد على عروض الكلب والفرس الصغير لتحسن الجودة التقنية لمشروعك. قد يكون للتحضير لها أثر غير مباشر في الجودة التقنية، لكن عادةً يُصرف وقت في صنع عرض شرائح جيد المنظر أكثر منه في تحسين جودة البرمجية. اعتمد على التفحصات أو العبورات أو قراءة الشفرة لتحسين الجودة التقنية.

مقارنة بين تقنيات البناء التعاوني

ما هي الفروق بين الأنواع المختلفة في البناء التعاوني؟ يقدم الجدول 21-1 ملخص عن المميزات الرئيسية لك تقنية.

جدول 21-1 مقارنة بين تقنيات البناء التعاوني

الخاصية	البرمجة الثنائية	التفحص الرسمي	المراجعة غير الرسمية (العبورات)
أدوار معروفة للمشتركين	نعم	نعم	لا
تدريب رسمي في كيفية أداء الأدوار	قد، بالتمرين	نعم	لا
من "يقود" التعاون	الشخص الذي يعمل على لوحة المفاتيح	الرئيس	الكاتب، عادةً
محور التعاون	التصميم والتشفير والاختبار وتصحيح الأخطاء	اكتشاف الأخطاء فقط	يتنوع
جهد المراجعة المركز—البحث عن أنواع الأخطاء الأكثر تكراراً بالإيجاد	رسمياً، إن كان الأمر بالمطلق	نعم	لا
متابعة لتقليل الإصلاحات السيئة	نعم	نعم	لا
أخطاء مستقبلية أقل بسبب التغذية الراجعة المفصلة عن الخطأ للمبرمجين المستقلين	بالصدفة	نعم	بالصدفة
فعالية محسنة للعملية من تحليل النتائج	لا	نعم	لا
مفيدة للنشاطات غير البنائية	من المحتمل	نعم	نعم
النسبة المئوية القياسية للأخطاء المكتشفة	40-60%	45-70%	20-40%

لا تمتلك البرمجة الثنائية عقود من التزويد ببيانات عن فعاليتها مثلما تمتلك التفحصات الرسمية، لكن البيانات الأولية توحي أنها والتفحصات على قدم المساواة تقريباً، والتقارير القصيرة المعتمدة على حالات فردية كانت إيجابية أيضاً.

إن كانت البرمجة الثنائية والتفحصات الرسمية تؤدي إلى نتائج متشابهة في الجودة والكلفة والمواعيد، فإن الاختيار بينهما يصبح مسألة متعلقة بالأسلوب الشخصي أكثر من للمادة التقنية. يفضل بعض الناس العمل بمفردهم، ونادراً ما يكسرون الأسلوب الفردي إلا لاجتماعات التفحصات فقط. يفضل آخرون أن يصرفوا وقتاً أكثر بالعمل مباشرة مع الآخرين. يمكن أن يُوجه الاختيار بين التقنيتين بواسطة الأساليب المفضلة للمطورين المحددين ضمن الفريق، وقد يسمح لمجموعات فرعية ضمن الفريق أن يختاروا الطريقة التي يحبون أن يعملوا بها معظم وقتهم. ينبغي أيضاً أن تستخدم تقنيات مختلفة في المشروع، حسبما يناسب.

مصادر إضافية

يوجد هنا مصادر أكثر، مهتمة بالبناء التعاوني[1].

البرمجة الثنائية

Williams, Laurie and Robert Kessler. Pair Programming Illuminated. Boston, MA: Addison Wesley, 2002. يشرح هذا الكتاب دواخل وخوارج البرمجة الثنائية. متضمناً كيفية معالجة الالتقاءات المتنوعة بين الشخصيات (مثلاً، الخبراء وغير الخبراء، والانطوائيين والمنفتحين...) وقضايا تطبيقية أخرى.

Beck, Kent. Extreme Programming Explained: Embrace Change. Reading, MA: Addison Wesley, 2000. يلامس هذا الكتاب البرمجة الثنائية برفق ويبين كيف يمكن أن تُستخدم مقترنةً مع التقنيات الداعمة لبعضها البعض الأخرى، متضمنةً معايير كتابة الشفرة والتكامل المتكرر والاختبار الرجعي.

Reifer, Donald. "How to Get the Most Out of Extreme Programming/Agile Methods," Proceedings, XP/Agile Universe 2002. New York, NY: Springer; pp. 185–196. تلخص هذه المقالة التجربة الصناعية **للبرمجة الزائدة** والمناهج **الرشيقة** وتقدم مفاتيح النجاح في البرمجة الثنائية.

[1] cc2e.com/2106

التفحصات

Wiegers, Karl. Peer Reviews in Software: A Practical Guide. Boston, MA: Addison-Wesley, 2002. يصف هذا الكتاب المكتوب بشكل جيد الدواخل والخوارج للأنواع المختلفة للمراجعات، متضمنة التفحصات الرسمية وغيرها، من التطبيقات الأقل رسمية. تمت الأبحاث فيه بشكل جيد، ولديه تركيز على التطبيق، وهو سهل القراءة.

Gilb, Tom and Dorothy Graham. Software Inspection. Wokingham, England: Addison-Wesley, 1993. هذا يحتوي نقاش معمق عن التفحصات بالقرب من بداية التسعينيات في القرن الماضي. لديه تركيز على التطبيق ويتضمن حالات دراسة تصف التجارب المتنوعة التي قامت بها المنظمات في إعداد برامج التفحص.

Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program Development." IBM Systems Journal 15, no. 3 (1976): 182–211.

Fagan, Michael E. "Advances in Software Inspections." IEEE Transactions on Software Engineering, SE-12, no. 7 (July 1986): 744–51. كُتبت هاتين المقالتين من قبل مطور التفحصات. إنهما تحتويان غذاء دسماً يحوي ما تحتاج معرفته لتجري تفحصاً، متضمناً كل صيغ التفحص المعيارية.

معايير ذات صلة

IEEE Std 1028-1997, Standard for Software Reviews، معيار لمراجعة البرمجيات

IEEE Std 730-2002, Standard for Software Quality Assurance Plans، معيار لمخططات ضمان جودة البرمجية.

نقاط مفتاحية

- تميل تطبيقات التطوير التعاوني لإيجاد أخطاء بنســب مئوية أعلى من الاختبار، وتميل لإيجادها بفعالية أكبر.
- تميل تطبيقات التطوير التعاوني لإيجاد أنواع مختلفة من الأخطاء بالمقارنة مع ما يجد الاختبار، وهذا يتضمن أنه عليك أن تستخدم كلا المراجعات والاختبار لتؤكد جودة برمجيتك.
- تستخدم التفحصات الرسمية لوائح اختبار وتحضير وقواعد معرّفة جيداً وعملية تحسين مستمرة لتعظيم فعالية اكتشاف الأخطاء. إنها تميل لإيجاد أخطاء أكثر من العبورات.
- تكلف البرمجة الثنائية عادةً نفس التفحص تقريباً وتؤدي إلى جودة شــفرة مشــابهة. البرمجة الثنائية قيّمة بشكل خاص عندما يكون تقصير جدول المواعيد مرغوباً. بعض المطورين يفضلون العمل بأزواج عن العمل بمفردهم.
- يمكن أن تُســتخدم التفحصــات الرســمية في منتجات العمل بشــكل مشــابه جداً لاســتخدامها في المتطلبات والتصميم وحالات الاختبار بالإضافة إلى الشفرة.
- العبورات وقراءة الشفرة بدلان للتفحصات. تؤمن قراءة الشفرة مرونة أكثر في ا ستخدام كل شخص لوقته بفعالية.

الفصل 22

اختبار المُطوّر

المحتويات[1]

مواضيع ذات صلة

- المنظر الطبيعي لجودة البرمجيات: الفصل 20
- ممارسات البناء التعاوني: الفصل 21
- التصحيح: الفصل 23
- التكامل: الفصل 29
- المتطلبات الأساسية لعميلة البناء: الفصل 3

إن الاختبار هو نشاط تحسين الجودة الأكثر شيوعاً- هو الممارسة المدعومة ماليًا من قبل البحوث الصناعية والأكاديمية والتجارب التجارية. تُختبر البرمجيات بعدة طُرق، يُنفذ بعضها بشكل نموذجي من قبل المطورين، ويُنفذ بعضها الآخر بشكل أكثر شيوعًا من قبل موظف الاختبار المُتخصص:

[1] cc2e.com/2261

- اختبار الوحدة (*Unit testing*) هو تنفيذ صف كامل، أو إجرائية، أو برنامج صغير مكتوب من قبل مُبرمج واحد أو فريق من المبرمجين، يُختبَر في معزل عن النظام الكامل.
- اختبار المكوّن (*Component testing*) هو تنفيذ صـــف، أو رزمة package، أو برنامج صــغير، أو عنصـــر برنامج آخر يتضـــمن عمل مجموعة من المبرمجين أو عدة فرق برمجة، يُختبر في معزل عن النظام الكامل.
- اختبار التكامل (*Integration testing*) هو تنفيذ مُشترك لأثنين أو أكثر من الصفوف، أو مكونات الرزم، أو النظم الفرعية المُنشـــأة من قبل عدة مبرمجين أو عدة فرق برمجة. يبدأ عادةً هذا النوع من الاختبار حالما يتواجد صفين للاختبار ويستمر حتى اكتمال النظام بأكمله.
- اختبار التراجع (*Regression testing*) هو تكرار حالات الاختبار الســـابقة لغرض العثور على عيوب في البرامج التي سبق أن اجتازت نفس مجموعة الاختبارات.
- اختبار النظام (*System testing*) هو تنفذ البرمجيات بنســـختها النهائية من الإعدادات، بما فيها التكامل مع البرمجيات والأجهزة الاخرى. حيث يختبر هذا الاختبار الأمن، والأداء، وخســـارة الموارد، ومشـــاكل التوقيت، والقضايا الأخرى، التي لا يمكن اختباراها عند مستويات أخفض من التكامل.

في هذا الفصل، تُشير كلمة "اختبار" إلى الاختبار من قبل المُطوّر، الذي يتكون عادةً من اختبارات الوحدة، واختبارات المكوّن، واختبارات التكامل، ولكن يمكن أن تتضمن أحيانًا اختبارات التراجع واختبارات النظام. تُنفذ الانواع العديدة الإضافية من الاختبار من قبل موظف اختبار مُتخصص ونادرًا ما يتم تنفيذها من قبل المطورين، بما في ذلك اختبارات بيتا (beta tests)، واختبارات استحسان الزبائن (customer-acceptance tests)، واختبارات الأداء (performance tests)، واختبارات التكوين (configuration tests)، واختبارات المنصّة (platform tests)، واختبارات الإجهاد (stress tests)، واختبارات قابلية الاستخدام (usability tests)، وإلى أخره. لم تتم مناقشة هذه الأنواع من الاختبارات في هذا الفصل.



يُقسم الاختبار عادةً إلى فئتين رئيسيتين: اختبار الصندوق الأسود واختبار الصندوق الأبيض (أو الصندوق الزجاجي). يُشير اختبار "الصندوق الأسود" إلى الاختبارات التي لا يمكن فيها أن يرى المُختبر الأعمال الداخلية للعنصر المُختبر. واضح أنه لا يُطبق هذا عندما تقوم باختبار الشفرة المكتوبة من قبلك. بينما يُشير "اختبار الصندوق الأبيض" إلى الاختبارات التي يكون فيها المُختبر مُدرك للأعمال الداخلية للعنصر المُختبر. وهذا هو نوع الاختبار الذي تستخدمه كمطوّر لاختبار الشفرة الخاصة بك. لكل من اختبار الصندوق الأسود واختبار الصندوق الأبيض نقاط ضعف ونقاط قوة؛ يُركز هذا الفصل على اختبار الصندوق الأبيض، لأنه نوع الاختبار الذي يقوم به المطورون.

يستخدم بعض المبرمجين مصطلحات "اختبار"، و"تصحيح"Debugging بشكل متبادل، ولكن يُميز المبرمجين الحريصين بين هذين النشاطين. الاختبار هو وسيلة الكشف عن الأخطاء. التصحيح هو وسيلة تشخيص وتصحيح الحالات الجذرية من الأخطاء، المكشوفة للتو. يتعامل هذا الفصل بشكل حصري مع كشف الخطأ. بينما يُناقَش تصحيح الخطأ في الفصل 23،"التصحيح".

إن قضية الاختبار الكاملة هي أكبر بكثير من موضوع الاختبار خلال عملية البناء. نُوقشت مواضيع اختبار النظام، واختبار الإجهاد، واختبار الصندوق الأسود، ومواضيع أخرى للمتخصصين في الاختبار، في القسم "مصادر إضافية" في نهاية هذا الفصل.

1.22 دور اختبار المُطوّر في جودة البرمجيات

إن الاختبار هو الجزء الأساسي في أي برنامج لجودة البرمجيات، وفي كثير من الأحيان هو الجزء الوحيد.[1] وهذا مؤسف، حيث ظهرت ممارسات التطوير التعاوني بصيغها المختلفة لإيجاد نسبة عالية من الأخطاء، أكثر ما يفعله الاختبار، وتكلف هذه الممارسات أقل من نصف التكلفة للعثور على كل خطأ أثناء الاختبار (كارد 1987، راسل 1991، كابلان 1995). عادةً تجد خطوات الاختبار المستقلة (اختبار الوحدة، واختبار المكون، واختبار التكامل) أقل من 50 بالمئة من الأخطاء الحالية لكل منها. بينما يجد عادةً مزيج خطوات الاختبار أقل من 60 بالمئة من الأخطاء الحالية (جونز 1998)

- إذا كنت ترغب بوضع قائمة بمجموعة نشاطات تطوير البرمجيات، وتسأل نفسك "أي من هذه الأشياء لا يشبه الأخرين؟" فسيكون الجواب هو "الاختبار"[2]. إن الاختبار هو نشاط صعب بالنسبة لمعظم المطورين، وذلك للأسباب التالية:
- يتعارض هدف الاختبار مع أهداف نشاطات التطوير الأخرى. إن الهدف هو إيجاد الأخطاء. الاختبار الناجح هو الذي يكسر البرمجية. بينما الهدف من كل نشاط تطوير آخر هو منع الأخطاء والمحافظة على البرمجية من الكسر(breaking).
- لا يمكن أبدًا للاختبار أن يثبت عدم وجود أخطاء. ففي حال قيامك بالاختبار بشكل موسّع، ووجدت آلاف الأخطاء، فهل يعني هذا أنك وجدت كل الأخطاء، أو لديك آلاف الأخطاء الأخرى لإيجادها؟ قد يعني غياب الخطأ عدم الفعالية أو حالات اختبار غير كاملة، بالسهولة نفسها التي يمكن فيها أن يعني غياب الخطأ برمجية مثالية.

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن موضوع المُراجعات، انظر الفصل 21 "البناء التعاوني".

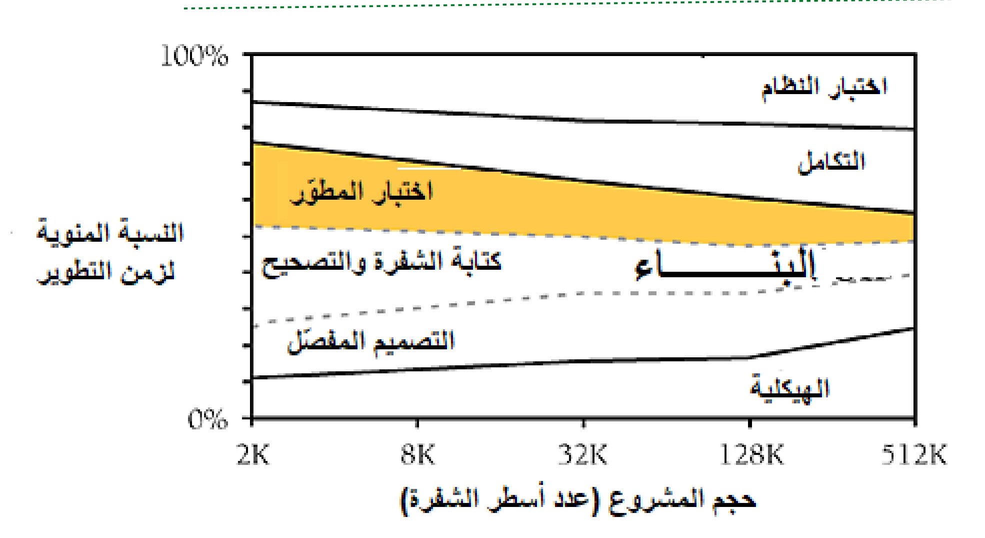
[2] لا تصاب البرامج بالأخطاء كما يصاب الناس بالجراثيم، بالحوام حول البرامج الأخرى الحاوية على أخطاء. بل يجب على المبرمجين إدخالها. - هارلان ميلز

- لا يثبت الاختبار وحده جودة البرمجية. إن نتائج الاختبار هي مؤشـــر على الجودة، ولكن في حد ذاتها لا تحسنها. إن محاولة تحسين جودة برمجية عن طريق زيادة كمية الاختبارات، هي مثل محاولة خسارة الوزن عن طريق وزن نفسك أكثر. سيحدد ما ستكون قد أكلته قبل الوقوف على ميزان القياس وزنك، وأيضًا ستحدد التقنيات التي تستخدمها لتطوير البرمجية، عدد الأخطاء التي سيجدها الاختبار. فإذا كنت ترغب بأن تفقد وزن، فلا تشـري مقياس وزن جديد؛ بل غيّر طريقة الحمية المُتبعة. وإذا كنت ترغب بتحسين البرمجية، فلا تختبرها أكثر؛ بل طوّرها بشكل أفضل.
- يطلب الاختبار منك افتراض أنك ستجد أخطاء في شـفرتك. وإذا افترضـت أنك لن تجد أخطاء، فلن تجدها، ولكن فقط لأنك قمت بالتنبؤ. فإذا نفذت برنامج مع أمل أنه لن يحتوي أية أخطاء، فســيكون من السهل التغا ضي عن الأخطاء التي تجدها. في درا سة أ صبحت كلا سيكية، حـصل جلينفورد مايرز ومجموعة من المبرمجين الخبراء على اختبار برنامج ب 15 عيب معروف. و جد المبرمج الوا حد بشكل متوسط فقط 5 من 15 خطأ. ووجد أفضل مبرمج 9 أخطاء. كان المصدر الرئيسي للأخطاء غير المُكتشـــفة هو عدم فحص المخرجات الخاطئة بعناية كافية. حيث كنت الأخطاء مرئية، ولكن لم يلاحظها المبرمجون (مايرز 1978).

يجب عليك أن تأمل إيجاد أخطاء في شـــفرتك. من الممكن أن يبدو أمل كهذا، تصـــرف غير طبيعي، ولكن عليك أن تأمل أنك أنت من سيجد الأخطاء وليس شخصاً آخر.

هنا سؤال أساسي وهو، كم من الوقت يجب أن يُصرف في اختبار المطور على مشروع نموذجي؟ إن الرقم الذي يتم الاستشهاد به بشكل شائع لجميع الاختبارات هو 50 بالمئة من الوقت المصروف على المشروع، ولكن هذا مضلل. أولًا، يضم هذا الرقم الاختبار والتصحيح؛ أي يأخذ الاختبار وقت أقل. ثانيًا، يُمثل هذا الرقم كمية الوقت المصروفة عادةً، وليس كمية الوقت التي يجب أن تُصرف. ثالثًا، يتضمن هذا الرقم اختبار مستقل كاختبار المطوّر.

كما يوضح الشكل 22-1، حسب حجم المشروع وتعقيده، يجب أن يستغرق اختبار المطوّر من 8 إلى 25 بالمئة من وقت المشروع الكلي. هذا يتفق مع الكثير من التقارير.



الشكل 22-1 كلما زاد حجم البرنامج، يستهلك اختبار المطور نسبة مئوية أقل من زمن التطوير الكلي. تم وصف تأثير حجم البرنامج بمزيد من التفاصيل في الفصل 27، "كيف يؤثر حجم البرنامج على عملية البناء".

السؤال الثاني هو، ماذا تفعل بنتائج اختبار المطور؟ يمكنك على الفور أن تستخدم النتائج للوصول إلى موثوقية المُنتج الموضوع تحت التطوير. حتى لو لم تُصحح أبدًا العيوب التي يجدها الاختبار، يصف الاختبار مدى موثوقية البرنامج. استخدام آخر للنتائج، يمكنهم تقديم دليل لتصحيحات البرنامج، وهذا ما يفعلونه عادةً. أخيرًا، مع مرور الوقت، يُساعد سجل العيوب الموجد من خلال الاختبار، على كشف أنواع الأخطاء الأكثر شيوعًا. تستطيع استخدام هذه المعلومات لاختيار دروس التدريب المُناسبة، وأنشطة المراجعة التقنية المستقبلية، وتصميم حالات الاختبار المستقبلية.

الاختبار خلال عملية البناء

في بعض الأحيان يتجاهل العالم الكبير الواسع موضوع هذا الفصل: اختبار "الصندوق الأبيض" أو "الصندوق الزجاجي". عمومًا، قد ترغب بتصميم صف ليكون صندوق أسود- حيث ليس على مستخدم الصف أن ينظر في ماضي الواجهة لمعرفة ماذا يفعل الصف. على كل حال، أثناء اختبار الصف، من المفيد التعامل معه كصندوق زجاجي، لرؤية شفرة المصدر الداخلية للصف، كما ورؤية كل مدخلاته وخرجه. فإذا كنت تعرف ما في داخل الصندوق، يمكنك اختبار الصف بشكل أكثر شمولية. بالطبع، سيكون لديك أيضًا النقاط العمياء نفسها في اختبار الصف، التي كانت لديك عن كتابة الصف، وبهذا لدى اختبار الصندوق الأسود حسنات أيضًا.

خلال عملية البناء، تكتب بشكل عام إجرائية أو صف، ومن ثم تتحقق منه عقليًا، ومن ثم تُراجعه أو تختبره. بغض النظر عن استراتيجية اختبار النظام أو التكامل المُستخدمة، عليك أن تختبر كل وحدة بعناية، قبل أن تجمعها مع الوحدات الأخرى. وإذا كنت تكتب عدة إجرائيات، فعليك اختبارهم كلهم مع بعض بنفس الوقت. ليست الإجرائيات حقًا سهلة الاختبار بشكل مستقل، ولكنها سهلة التصحيح بشكل مستقل. إذا رميت عدة إجرائيات غير مختبرة مع بعضها مرة واحدة ووجدت خطأ، فقد تكون أي واحدة من هذه الإجرائيات هي المسؤولة. وإذا أضفت إجرائية واحدة إلى مجموعة من الإجرائيات المُختبرة سابقًا، عندها ستعرف أن أيّة أخطاء

جديدة هي نتيجة الإجرائية الجديدة أو بسبب التفاعلات مع الإجرائية الجديدة. مهمة التصحيح أسهل.

لدى ممارسات البناء التعاوني العديد من نقاط القوة، التي لا يمكن للاختبار أن يُطابقها. ولكن جزء من المشكلة مع الاختبار، أنه لا يتم إنجاز الاختبار غالبًا كما يجب أن يُنجز. يستطيع المطور إنجاز مئات الاختبارات، ولا يختبر عندها إلا جزء من الشفرة. لا يعني الشعور بتغطية جيدة للاختبار، أن تغطية الاختبار الفعلية كافية بالحقيقة. يمكن أن يدعم فهم مفاهيم الاختبار الأساسية اختبار أفضل ويرفع فعالية الاختبار.

2.22 النهج الموصى به لاختبار المُطوّر

يزيد النهج المنظم لاختبار المطور قدرتك على كشف الأخطاء من كل الأنواع بأقل جهد ممكن. تأكد من تغطية هذه الأساسيات:

- اختبر كل المتطلبات ذات الصلة لتتأكد من تنفيذ جميع المتطلبات. خطط واختبر الحالات لكل خطوة في مرحلة المتطلبات، أو في أقرب وقت ممكن- من المفضل قبل البدء بكتابة الوحدة المُختبرة. فكّر في اختبار الإغفال الشائع للمتطلبات. إن مستوى الأمن، والتخزين، وإجراء التهيئة، وموثوقية النظام، هي كلها لعبة عادلة للاختبار وغالبًا تُغفل في مرحلة المتطلبات.
- اختبر كل زاوية تصميم ذات صلة لتتأكد من تنفيذ التصميم. خطط كل حالات الاختبار لهذه الخطوة في مرحلة التصميم، أو في أقرب وقت ممكن- قبل البدء بكتابة الشفرة المفصلة للإجرائية أو للصف المُختبر.
- استخدم "اختبار الأساس" لإضافة حالات الاختبار المفصّلة إلى تلك التي تختبر المتطلبات والتصميم. أضف اختبارات تدفق المعطيات، ومن ثم أضف حالات الاختبار المتبقية الضرورية لتنفيذ الشفرة بشكل تام. على الأقل، عليك اختبار كل سطر من الشفرة. إن اختبار الأساس واختبار تدفق المعلومات موصوفين لاحقًا في هذا الفصل.
- استخدم قائمة التحقق لأنواع الأخطاء التي قمت بها في المشروع حتى الآن، أو قمت بها في مشاريع سابقة.

صمم حالات الاختبار مع المنتج. يمكن أن يساعدك هذا بتجنّب الأخطاء في المتطلبات والتصميم، التي تميل إلى أن تكون أكثر تكلفة من الأخطاء في كتابة الشفرة. خطط للاختبار وإيجاد عيب بشكل أبكر قدر الإمكان، لأنه أكل تكلفة إصلاح العيوب بشكل باكر.

اختبر أولًا أو أخيرًا؟

أحيانًا يتساءل المطورين فيما إذا كان من الأفضل كتابة حالات الاختبار بعد كتابة الشفرة أو مُسبقًا (بيك 2003). يقترح الرسم البياني حول زيادة التكلفة المتعلقة بالعيب - انظر الشكل 3-1 في الصفحة- أن كتابة حالات الاختبار أولًا سينقص المدة الزمنية بين وقت إدخال العيب في الشفرة وبين وقت الكشف عن العيب وإزالته. يتبين أن هذه واحدة من عدة أسباب لكتابة حالات الاختبار أولًا:

- إن كتابة حالات الاختبار قبل كتابة الشــفرة لا يأخذ جهد أكبر من حالة كتابة حالات الاختبار بعد كتابة الشفرة؛ هذا ببساطة تغيير لترتيب القيام بنشاط كتابة حالات الاختبار.
- عندما تكتب حالات الاختبار أولًا، تكتشف العيوب أبكر وتستطيع إصلاحهم بشكل أكثر سهولة.
- تجبرك كتابة حالات الاختبار أولًا على التفكير على الأقل قليلًا حول المتطلبات والتصــميم قبل كتابة الشفرة، الذي يؤدي إلى إنتاج شفرة أفضل.
- كتابة حالات الاختبار أولًا تكشــف باكرًا عن مشــاكل المتطلبات، وذلك قبل كتابة الشــفرة، ذلك لأنه من الصعب كتابة حالة اختبار لمتطلب ضعيف.
- إذا قمت بحفظ حالات الاختبار، وهذا ما يجب عليك القيام به- فلاتزال تســتطيع أن تختبر أخيرًا، بالإضافة إلى الاختبار أولًا.

أعتقد أن البرمجة مع الاختبار أولًا هي من أكثر الممارسات البرمجية المُفيدة التي ظهرت في العقد الماضي، وهي نهج عام جيد. ولكنه ليس بترياق الاختبار، لأنه يخضع للقيود العامة لاختبار المطور، المشروحة في القسم التالي.

قيود اختبار المطور

راقب القيود التالية لاختبار المطور:

تميل اختبارات المطور إلى أن تكون "اختبارات نظيفة". يميل المطورون إلى اختبار فيما إذا كانت الشفرة تعمل (اختبارات نظيفة)، بدلًا من اختبار كل الطرق التي تفشل فيها الشفرة (اختبارات غير نظيفة). تميل منظمات الاختبار غير الناضجة إلى امتلاك خمسة اختبارات نظيفة، مقابل كل اختبار غير نظيف. أما منظمات الاختبار الناضجة فتميل إلى امتلاك خمسة اختبارات غير نظيفة، مقابل كل اختبار نظيف. لا يتم عكس هذه النسبة عن طريق تقليل الاختبارات النظيفة؛ بل يتم ذلك عن طريق إنشاء 25 مرة أكثر من هذه الاختبارات غير النظيفة (بوريس بييزر في جونسون 1994).



يميل اختبار المطوِّر إلى امتلاك نظرة تفاؤلية عن منطقة تغطية الاختبار يعتقد عدد متوسط من المبرمجين بأنهم ينجزون نسبة 95 بالمئة من منطقة تغطية الاختبار، ولكنهم بالعادة ينجزون، في

أحسن الأحوال، أكثر من نسبة 80 بالمئة من منطقة تغطية الاختبار، وينجزون نسبة 30 بالمئة في أسوء الأحوال، وبين 50 و60 بالمئة في الحالات المتوسطة (بوريس بييزر في جونسون 1994).

يميل اختبار المطوِّر إلى أن يتجاوز الأنواع الأكثر تطورًا من منطقة التغطية. يعتبر معظم المطورين نوع تغطية الاختبار المعروف بـ "تغطية العبارات بنسبة 100%"، نوعا كافيا. هذه بداية جيدة، لكنها بالكاد كافية. إن معيار التغطية الأفضل هو بمقابلة ما يُدعى "تغطية الفرع بنسبة 100%"، مع اختبار كل مصطلح على الأقل لقيمة صحيحة واحدة وقيمة خاطئة واحدة. يوفر القسم 3.22، "مجموعة من حيل الاختبار" المزيد من التفاصيل حول كيفية إنجاز هذا.

لا تقلل أي من هذه النقاط من قيمة اختبار المطوِّر، ولكنها تقدم مساعدة في وضع اختبار المطوِّر في المنظور المناسب. على الرغم من قيمة اختبار المطوِّر، فهو غير كافٍ بمفرده لتأمين ضمان الجودة الكافية، ويجب أن يتم إضافة إليه عدة ممارسات أخرى، بما فيها الاختبار المستقل، وتقنيات البناء التعاونية.

3.22 مجموعة من حيل الاختبار

لماذا لا يمكن اثبات صحة برنامج من خلال اختباره؟ لاستخدام الاختبار لإثبات أن البرنامج يعمل، عليك اختبار كل قيمة دخل ممكنه للبرنامج، وكل مجموعات قيم الدخل الممكنة. وحتى من أجل المشاريع الصغيرة، مثل هكذا تعهد سوف يصبح باهظ للغاية. افترض على سبيل المثال، لديك برنامج يأخذ كدخل اسم، وعنوان، ورقم الهاتف، ويخزنهم في ملف. هذا بالتأكيد برنامج بسيط، أبسط بكثير من أية برامج يجب عليك القلق حول صحتها. افترض أن طول أي من الأسماء الممكنة هو 20 محرف، ويوجد 26 محرف لاستخدامها في هذه الأسماء. هذا ما سيكون عدد المدخلات الممكنة للبرنامج:

الاسم	26^{20} (20 محرف، كل واحد منها بـ 26 اختيار)
العنوان	26^{20} (20 محرف، كل واحد منها بـ 26 اختيار)
رقم الهاتف	10^{10} (10 أرقام، كل منها بـ 10 اختيارات)
الاحتمالات الكلية	$26^{20}*26^{20}*10^{10} \approx 10^{66}$

حتى لو كمية الدخل صغيرة نسبيًا، لديك 10^{66} حالة اختبار ممكنة. أي إذا قام أحدهم باختبار هذا البرنامج بمعدل ترليون حالة اختبار بالثانية، فسينجز باليوم الواحد فقط 1 بالمئة. لاحظ، في حال إضافتك لكمية أكثر واقعية من البيانات، ستصبح مهمة اختبار كل الحالات غير ممكنه بشكل أكبر.

اختبار غير مكتمل

بما أن الاختبار الشامل (كل حالات الاختبار) غير ممكن[1]، من الناحية العملية، أحد الخيارات هو باختبار أكثر حالات الاختبار التي من الممكن ان تجد أخطاء. من 10^{66} حالة اختبار ممكنة، يوجد فقط عدد قليل منها من المحتمل أن تكتشف أخطاء، والآخرى لا يمكنها ذلك. عليك أن تركز على مجموعة حالات الاختبار التي تخبرك أشياء مختلفة، بدلًا من مجموعة أخرى تخبرك الشيء نفسه تكرارًا وتكرارًا.

عندما تخطط الاختبارات، احذف تلك التي لا تخبرك أي شيء جديد- مثل الاختبارات التي من الممكن ألا تنتج أخطاء مع أنواع جديدة من المعطيات، إذا كانت الأنواع المشابهة لها من المعطيات لا تنتج أخطاء. اقترح أشخاص مختلفين مناهج مختلفة لتغطية الأساسيات بشكل فعّال، وسنُناقش العديد من هذه المناهج في الأقسام التالية.

اختبار الأساس المنظم

بالرغم من الاسم المعقد، إن اختبار الأساس المُنظم هو مفهوم بسيط. الفكرة أنه عليك اختبار كل عبارة برمجية في برنامج على الأقل مرة واحدة. إذا كانت العبارة هي عبارة منطقية- على سبيل المثال عبارة `if`، أو عبارة `while`- عليك أن تغير الاختبار وذلك حسب تعقيد التعبير داخل `if`، أو `while`، وذلك للتأكد من أنه يتم اختبار العبارة بالكامل. أسهل طريقة للتأكد من تغطيتك لكل الأساسيات، هي بحساب عدد المسارات خلال البرنامج ومن ثم تطوير العدد الأصغر من حالات الاختبار التي ستختبر كل مسار خلال البرنامج.

من الممكن أنك قد سمعت عن اختبار "منطقة تغطية الشفرة" أو اختبار "منطقة التغطية المنطقية". هما متقاربين من حيث اختبارك لكل المسارات في برنامج. بما أنهم يغطون كل المسارات، فهما شبيهين مع اختبار الأساس المنظم، ولكنهم لا يتضمنان فكرة تغطية كل المسارات مع مجوعة *صغيرة* من حالات الاختبار. في حال استخدامك لاختبار منطقة تغطية الشفرة أو اختبار منطقة التغطية المنطقية، يمكنك إنشاء العديد من حالات الاختبار، أكثر مما تحتاج له في حالة التغطية المنطقية نفسها باستخدام اختبار الأساس المنظم.

يمكن حساب العدد الأصغري لحالات الاختبار التي تحتاجها لاختبار الأساس،[2] كما في الطريقة التالية:

1. ابدأ بالعدد 1 من أجل المسار المستقيم خلال الإجرائية.

[1] **إشارة مرجعية**: واحدة من طرق إخبارك فيما إذا قد تمت تغطية كل الشفرة، هي باستخدام مراقب التغطية. لمزيد من التفاصيل، انظر "مراقبات التغطية" في القسم 5.22، "أدوات دعم الاختبار"، في هذا الفصل.

[2] **إشارة مرجعية:** هذه الإجرائية مُشابهة لتلك التي تقيس التعقيد في "كيفية قياس التعقيد" في القسم " 6.19.

2. أضف العدد 1 في حال مصادفتك أي من الكلمات المفتاحية التالية، أو مكافئتها: `if`، و`while`، و`repeat`، و`for`، و`and`، و`or`.

3. أضف العدد 1 في حالة عبارة الحالة case. وإذا لم تكن لدى عبارة الحالة ، حالة افتراضية، أضف أيضًا 1.

فيما يلي مثال:

مثال بسيط عن حساب عدد المسارات خلال برنامج جافا.

```
Statement1;
Statement2;          <----  عد "1" من أجل الإجرائية بحد ذاتها.
if ( x < 10 ) {
  Statement3;        <----  عد "2" من أجل عبارة if.
}
Statement4;
```

في هذا المثال، تبدأ بالعدد 1، ومن ثم تضيف 1 لعبارة `if`، ليصبح المجموع 2. هذا يعني انه عليك أن تملك على الأقل حالتي اختبار لتغطية كل المسارات خلال البرنامج. في هذا المثال، عليك أن تملك حالات الاختبار التالية:

- تنفيذ العبارات المتحكم بها من قبل if (x < 10).
- عدم تنفيذ العبارات المتحكم بها من قبل if (x >= 10).

يحتاج مثال الشفرة هذه إلى أن يكون مثال حقيقي، لإعطاء فكرة دقيقة عن كيفية عمل هذا النوع من الاختبار. تشمل الواقعية في هذه الحالة شفرة تحتوي على عيوب.



القائمة التالية هي مثال أكثر تعقيدًا بشكل قليل. قطعة الشفرة هذه مُستخدمة على طول الفصل وتحتوي على القليل من الأخطاء الممكنة.

مثال عن حساب عدد الحالات المُحتاجة لاختبار الأساس على برنامج جافا.

```
1 // حساب الآجر الصافي
2 totalWithholdings = 0;
3
4 for ( id = 0; id < numEmployees; id++ ) {
5
6  // حساب حسم الضمان الاجتماعي، إذا كان أقل من الحد الأقصى
7   if ( m_employee[ id ].governmentRetirementWithheld <
MAX_GOVT_RETIREMENT ) {
8    governmentRetirement = ComputeGovernmentRetirement( m_employee[ id
] );
9   }
10
11  // جعل الافتراضي لا مساهمة للتقاعد
12  companyRetirement = 0;
13
14  // تحديد مساهمة التقاعد التقديرية للموظفين
15  if ( m_employee[ id ].WantsRetirement &&
16    EligibleForRetirement( m_employee[ id ] ) ) {
17    companyRetirement = GetRetirement( m_employee[ id ] );
18  }
19
20  grossPay = ComputeGrossPay ( m_employee[ id ] );
21
22  // IRA تحديد مساهمة
23  personalRetirement = 0;
24  if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
25    personalRetirement = PersonalRetirementContribution( m_employee[
id ],
26     companyRetirement, grossPay );
27  }
28
29  // اجعل شيك الراتب اسبوعي
30  withholding = ComputeWithholding( m_employee[ id ] );
31  netPay = grossPay - withholding - companyRetirement -
governmentRetirement –
32    personalRetirement;
33  PayEmployee( m_employee[ id ], netPay );
34
```

عد "1" للإجرائية نفسها.

عد "2" للعبارة for.

عد "3" للعبارة if.

عد "4" من أجل if و"5" من أجل &&.

عد "6" للعبارة if.

```
35  // أضف راتب الموظف هذا إلى الإجمالي للمحاسبة
36  totalWithholdings = totalWithholdings + withholding;
37  totalGovernmentRetirement = totalGovernmentRetirement +
governmentRetirement;
38  totalRetirement = totalRetirement + companyRetirement;
39 }
40
41 SavePayRecords( totalWithholdings, totalGovernmentRetirement,
totalRetirement );
```

في هذا المثال، ستحتاج إلى حالة اختبار واحدة مبدئية، بالإضافة إلى واحدة لكل من الكلمات المفتاحية الخمسة، بمجموع كلي يساوي 6 عبارات. هذا لا يعني أنه أية حالات اختبار ستة ستغطي كل الأساسات. بل يعني، على الأقل، مطلوب ست حالات اختبار. إذا لم يتم بناء الحالات بعناية، فمن شبه المؤكد أنها لن تغطي جميع الأساسات. الخدعة هي بالانتباه إلى نفس الكلمات المفتاحية التي تستخدمها، عند حساب عدد الحالات المحتاجة. تمثل كل كلمة مفتاحية في الشفرة شيء ما، من الممكن أن يكون صحيح أو خاطئ؛ تأكد من امتلاكك على الأقل حالة اختبار واحدة لكل حالة صحيحة، وحالة اختبار واحدة على الأقل لكل حالة خاطئة.

فيما يلي مجموعة من حالات الاختبار، التي تغطي كل الأساسيات في هذا المثال:

الحالة	وصف الاختبار	معطيات الاختبار
1	الحالة الإسمية	كل الشروط المنطقية (البوليانية) صحيحة
2	الحالة البدائية للشرط هي false	numEmployees < 1
3	أول عبارة if هي false	m_employee[id].governmentRetirementWithheld>=MAX_GOVT_RETIREMENT
4	ثاني عبارة if هي false لأن الجزء الأول من and هو false	not m_employee[id].WantsRetirement
5	ثاني عبارة if هي false لأن الجزء الثاني من and هو false	not EligibleForRetirement(m_employee[id])
6	ثالث عبارة if هي false	not EligibleForRetirement(m_employee[id])

ملاحظة: سيتم في هذا الفصل توسيع هذا الجدول بحالات اختبار إضافية.

إذا كانت الإجرائية أكثر تعقيدًا من هذه، سيزداد بشكل سريع عدد حالات الاختبار لاستخدامها فقط لتغطية كل المسارات. تميل الإجرائيات القصيرة لامتلاك مسارات أقل لاختبارها. التعابير المنطقية بدون عدد كبير من and و or لديها اختلافات أقل للاختبار. إن سهولة الاختبار هي سبب إضافي جيد آخر للمحافظة على الإجرائيات قصيرة وعلى التعابير المنطقية بسيطة.

الآن بما أنك قد أنشأت ست حالات اختبار من أجل إجرائية، وحققت متطلبات اختبار الأساس المنظم، فهل تستطيع أخذ بالاعتبار الإجرائية ليتم اختبارها بشكل تام؟ من الممكن لا.

يضمن لك هذا النوع من الاختبار فقط أن الشفرة كلها ستُنفذ. وهو لا يأخذ بالحسبان تغيير المعطيات.

اختبار تدفق المعطيات

يعطيك هذه القسم والقسم الأخير مع بعضهم مثال توضيحي آخر عن أن تدفق التحكم وتدفق المعطيات متساويين بالأهمية في برمجة الحاسوب.

يستند اختبار تدفق المعطيات إلى فكرة أن استخدام المعطيات على الأقل هو معرض للأخطاء كما هو متحكم بالتدفق. يدّعي ***بوريس بيزير*** أنه على الأقل نصف الشفرة تحوي على عمليات تصريح للمعطيات وعمليات تهيئة (بيزير 1990).

من الممكن أن تنوجد المعطيات في واحدة من الحالات الثلاثة:

- **مُعرّفة (Defined):** تم تهيئة المعطيات، ولكن لم يتم استخدامها بعد.
- **مُستخدمة (Used):** تم استخدام المعطيات في عمليات الحساب، أو كمُعامل لإجرائية، أو لشيء آخر.
- **مقتولة (Killed):** عُرّفت المعطيات مرة واحدة، ولكن تم عدم تعريفها بطريقة ما. على سبيل المثال، إذا كانت المعطيات مؤشر، من الممكن أنه تم حرق المؤشر. إذا كان على سبيل المثال، فهرس لحلقة for، من الممكن أن يكون البرنامج خارج الحلقة ولا تعرف لغة البرمجة قيمة فهرس حلقة for، إذا كان خارج الحلقة. وإذا كان مؤشر لسجل في ملف، من الممكن أنه تم إغلاق الملف ولم يعد المؤشر للسجل مُتاح.

بالإضافة إلى المصطلحات "مُعرّف"، و "مُستخدم"، و"مقتول"، من المنطقي أن يوجد مصطلحات تصف الداخل والخارج من الإجرائية بشكل مباشر، قبل أو بعد القيام بشيء ما لمتغير:

- **المُدخل (Entered):** يدخل تدفق التحكم الإجرائية مباشرةً قبل أن يتم التصرف بالمتغير. على سبيل المثال، يُهيئ المتغير العامل في قمة إجرائية.
- **المُخرج (Exited):** يُغادر تدفق التحكم مباشرة الإجرائية بعد أن يتم التصرف بالمتغير. على سبيل المثال، يتم إسناد القيمة المُعادة إلى متغير حالة في نهاية الإجرائية.
-

تركيبات من حالات المتغيرات

التركيب الطبيعي من حالات المعطيات، هي أن يكون المتغير مُعرف ومُستخدم مرة أو أكثر من مرة، ومن الممكن أن يكون مقتول. انظر إلى النماذج التالية بارتياب:

- **مُعرّف - مُعرّف (Defined-Defined):** إذا كان عليك تعريف متغير مرتين قبل الاســتقرار على قيمة، أنت لا تحتاج برنامج أفضل، بل تحتاج إلى حاسوب أفضل! إنها عملية مهدرة ومعرضة للخطأ، حتى وإن لم تكن خاطئة في الواقع.
- **مُعرّف - مُخرج (Defined-Exited):** إذا كان المتغير محلي، فلا معنى لتعريفه والخروج دون ا ستخدامه. أما إذا كان وسيط لإجرائية أو متغير عام، فمن الممكن أن يكون هذا صحيح.
- **مُعرّف - مقتول (Defined-Killed):** إن تعريف مُتغير ومن ثم قتلـه يقترح إن المتغير إمـا خـارجي (غريب) أو غياب الشفرة التي من المفروض أن تستخدم هذا المتغير.
- **المُدخل – المقتول (Entered-Killed):** هذه مشــكلة إذا كان المتغير محلي. فلن يكون بحاجة إلى أن يكون مقتول، إذا تم تعريفه أو استخدامه. وإذا بالحالة المعاكسة، كان المتغير وسيط إجرائية أو متغير عام، هذا النوع صحيح طالما المتغير مُعرف في مكان ما قبل أن يتم قتله.
- **مُدخل – مُســتخدم مرة ثانية (Entered-Used Again):** هذا مشــكلة إذا كان المتغير محلي. يحتاج المتغير إلى أن يكون مُعرّف قبل اســتخدامه. وإذا بالحالة المعاكســة، كان المتغير وســيط إجرائية أو متغير عام، هذا النوع صحيح طالما المتغير مُعرف في مكان ما قبل أن يتم استخدامه.
- **مقتول – مقتول (Killed-Killed):** لا يجب أن يحتاج المُتغير إلى أن يُقتل مرتين. وأيضًــا القتل مرتين مُهلك للمؤشرات- واحدة من أفضل الطرق لتعليق جهازك هو قتل (بشكل مجاني) مؤشر مرتين.
- **مقتول – مستخدَم (Killed-Used):** استخدام متغير بعد قتله هو خطأ منطقي. إذا بدى أن الشفرة تعمل في أي مكان (على ســبيل المثال، المؤشــر الذي لا يزال يؤشــر إلى ذاكرة تم تحريرها). هذا حادث غير مقصود، يقول قانون مارفي أن الشفرة ستتوقف عن العمل في وقت التي ستسبب فيه فوضى كبيرة.
- **المُستخدم – المُعرَّف (Used-Defined):** استخدام ومن ثم تعريف مُتغير من الممكن أن لا يكون مشكلة، وذلك حســب إذا تم تعريف المتغير أيضًـا قبل اســتخدامه. بشـكل مؤكد إذا رأيت مخطط مُسـتخدم – مُعرف، من المفيد التحقق من تعريف سابق للمتغير.

تحقق من وجود هذه التتابعات غير الطبيعية لحالات البيانات قبل بدء الاختبار. بعد تحققك من التتابعات غير الطبيعية، أساس كتابة حالات اختبار تدفق المعطيات هو بإجراء كل المسارات مُعرف-مُستخدم الممكنة. يمكنك القيام بذلك بدرجات مختلفة من الدقة، بما في ذلك

- كل التعريفات. اختبر كل تعريف لكل متغير- أي في كل مكان يســتقبل فيه أي متغير قيمة. هذه استراتيجية ضعيفة لأنه إذا حاولت تطبيق هذا على كل سطر من الشفرة، ستفعل هذا بشكل افتراضي.

- كل التراكيب مُعرف – مُستخدم. اختبر كل تراكيب التعريفات لمتغير في مكان واحد، واستخدامهم في مكان آخر. هذه استراتيجية أقوى من اختبار كل التعريفات، لأن تنفيذ فقط كل سطر من الشفرة لا يضمن اختبار كل تركيبة مُعرف – مُستخدم.

فيما يلي مثال:

مثال بلغة البرمجة جافا لبرنامج، تُختبر له تدفق المعطيات.

```
if ( Condition 1 ) {
  x = a;
}
else {
  x = b;
}
if ( Condition 2 ) {
  y = x + 1;
}
else {
  y = x - 1;
}
```

لتغطية كل مسار في البرنامج، تحتاج لحالة اختبار واحدة في حالة الشرط الأول `Condition 1` صحيح، وحالة اختبار أخرى في حالة الشرط الأول خاطئ. أيضًا تحتاج لحالة اختبار واحدة في حالة الشرط الثاني `Condition 2` صحيح، وحالة اختبار أخرى في حالة الشرط الثاني خاطئ. يمكن التعامل مع هذا عن طريق حالتين اختبار: الحالة الأولى (الشرط 1 = صحيح، الشرط 2=صحيح)، والحالة الثانية (الشرط 1 = خطأ، الشرط 2 = خطأ). حالتي الاختبار هذين هي كل ما تحتاج إليه لاختبار الأساس المُنظم. وهي كل ما تحتاج إليه لتنفيذ كل سطر من الشفرة المُعرّفة لمتغير؛ حيث يعطوك بشكل أتوماتيكي الصيغة الضعيفة من اختبار تدفق المعطيات.

لتغطية كل تركيب مُعرف-مُستخدم، على كل حال، تحتاج لإضافة القليل من حالات الاختبار الإضافية. حتى الآن لدك الحالات المُنشئة عندما الشرط 1 والشرط 2 صحيحين في الوقت نفسه، وعندما الشرط 1 والشرط 2 خاطئين في الوقت نفسه:

```
x = a
...
y = x + 1
```

و

```
x = b
...
y = x - 1
```

ولكن تحتاج إلى حالتي اختبار إضافيتين لاختبار كل تركيب مُعرف – مُستخدم:

(1) x = a ومن ثم y = x – 1

(2) x = b ومن ثم y = x + 1

في هذا المثال، يمكنك الحصول على هذه التراكيب بإضافة حالتين إضافتين: الحالة 3 (الشرط 1= صحيح، الشرط 2= خطأ)، والحالة 4 (الشرط 1 =خطأ، الشرط 2= صحيح).

هناك طريقة جيدة لتطوير حالات اختبار وهي بالبدء باختبار الأساس المُنظم، الذي يعطيك القليل إذا لم يكن كل شيء من تدفقات المعطيات مُعرف-مُستخدم. ومن ثم أضف الحالات التي لا تزال تحتاجها للحصول على مجموعة كاملة من حالات اختبار تدفق المعطيات مُعرف- مُستخدم.

كما نُوقش في هذا الفصل، يؤمن اختبار الأساس المُنظم ست حالات اختبار للإجرائية التي تبدأ عند الصفحة 740. يتطلب اختبار تدفق المعطيات لكل زوج مُعرف- مُستخدم العديد من حالات الاختبار الأخرى، تمت تغطية بعضها في حالات الاختبار الحالية وبعضها لم يتم تغطيته. فيما يلي تراكيب تدفق المعطيات التي تُضيف حالات اختبار، غير تلك التي تم توليدها باختبار الأساس المُنظم:

الحالة	وصف الاختبار
7	تعريف companyRetirement في السطر 12، واستخدمه أول مرة في السطر 26. لا يُغطي هذا بالضرورة أي من حالات الاختبار السابقة.
8	تعريف companyRetirement في السطر 12، واستخدمه أول مرة في السطر 31. لا يُغطي هذا بالضرورة أي من حالات الاختبار السابقة.
9	تعريف companyRetirement في السطر 17، واستخدمه أول مرة في السطر 31. لا يُغطي هذا بالضرورة أي من حالات الاختبار السابقة.

بمجرد أن تمر عبر عملية سرد حالات اختبار تدفق المعطيات عدة مرات، ستتعرف على الحالات المثمرة وتلك التي تمت تغطيتها بالفعل. عندما تنتهي، ضع بقائمة كل التراكيب مُعرف- مُستخدم. قد يبدو هذا عمل كبير، ولكنه يكفل أن يظهر أية حالات لم تختبرها بشكل مجاني في منهج اختبار الأساس.

تقسيم متكافئ

تُغطي حالة الاختبار الجيدة جزء كبير من معطيات الدخل الممكنة.[1] إذا كشفت حالتي اختبار تمامًا عن نفس الأخطاء، عندها تحتاج إلى حالة اختبار واحدة فقط منهم. إن مفهوم "التقسيم المُتكافئ" هو صياغة لهذه الفكرة، ويساعد على إنقاص عدد حالات الاختبار المطلوبة.

في المثال الذي يبدأ في الصفحة 740، السطر 7 هو مكان جيد لاستخدام التقسيم المتكافئ. الشرط الذي يتم اختباره هو

[1] **إشارة مرجعية**: نُقش التقسيم المُتكافئ بشكل أكثر عمقًا في الكتب الموجودة في قسم "المراجع الإضافية" في نهاية هذا الفصل.

```
m_employee[ ID ].governmentRetirementWithheld <
MAX_GOVT_RETIREMENT.
```

لدى هذه الحالة صفين مُتكافئين: الصف الذي فيه

```
m_employee[ ID ].governmentRetirementWithheld
```

أصغر من MAX_GOVT_RETIREMENT

والصف الذي فيه أكبر أو يساوي MAX_GOVT_RETIREMENT.

من الممكن أن يكون لدى الأجزاء الاخرى من البرنامج صفوف متكافئة ذات صلة، التي تعني أنك تحتاج إلى اختبار أكثر من قيمتين ممكنتين لـ

```
m_employee[ ID ].governmentRetirementWithheld
```

ولكن بقدر ما يتعلق الأمر بهذا الجزء من البرنامج، فهناك حاجة إلى اثنين فقط.

لن يعطيك التفكير بالتقسيم المُتكافئ الكثير من الأفكار الجديدة في البرنامج، عندما تكون قد غطيت للتو البرنامج باستخدام اختبار الأساس واختبار تدفق المعطيات. على كل حال، إن هذا مُساعد بشكل خاص عندما تنظر إلى برنامج من الخارج (من المواصفات بدلًا من شفرة المصدر)، أو عندما المعطيات معقدة، ولا يتم عكس التعقيدات كلها في منطق البرنامج.

تخمين الخطأ

بالإضافة إلى تقنيات الاختبار الرسمية، المبرمجون الجيدون يستخدمون مجموعة مختلفة من تقنيات استدلال غير رسمية للكشف عن الأخطاء في شفرتهم[1]. واحده من هذه التقنيات هي تقنية تخمين الخطأ. إن مصطلح "تخمين الخطأ" هو اسم متواضع لمفهوم حساس. يعني هذا المفهوم إنشاء حالات اختبار بالاعتماد على تخمينات حول المكان الممكن لحدوث الخطأ في البرنامج، على الرغم من أنه ينطوي على قدر معين من التطور في التخمين.

يمكنك أن تحزر بشكل أساسي بالاعتماد على الحدس أو على تجارب قديمة. يُشير الفصل 21، "عملية البناء التعاونية" إلى فضيلة واحدة من عمليات التفتيش، أنهم ينتجون ويحافظون على قائمة بالأخطاء المشتركة. تُستخدم هذه القائمة للتحقق من شفرة جديدة. عندما تحتفظ بسجلات لأنواع الأخطاء التي قد ارتكبتها قبل، تحسن احتمال اكتشاف تقنية "تخمين الخطأ" للخطأ.

تصف الأقسام التالية القليلة أنواع خاصة من الأخطاء التي تصلح لتخمين الخطأ.

[1] **إشارة مرجعية:** لمزيد من التفاصيل عن الاستدلال، انظر القسم 2.2 "كيفية استخدام استعارات البرمجيات".

تحليل الحدود

واحدة من أكثر المناطق المثمرة للاختبار هي شروط الحدود – خطأ بعيدا من واحدة "خطأ الخطوة الواحدة" (off-by-one errors). أي قول num – 1 عندما تعني num، وقول => عندما تعني > هي أخطاء شائعة.

الفكرة من تحليل الحدود هو بكتابة حالات اختبار تُنفذ شروط الحدود. بشكل بياني، إذا كنت تختبر مجال من القيم التي هي أقل من max، فسيكون لديك ثلاث شروط ممكنة:



كما يظهر، يوجد ثلاث حالات للحدود: أقل تمامًا من max، وmax نفسها، وأكبر تمامًا من max. يتطلب الأمر ثلاث حالات اختبار لضمان عدم وقوع أي من الأخطاء الشائعة.

تحتوي عينة الشفرة في الصفحة 740 على فحص لـ

```
m_employee[ ID ].governmentReirementWithheld < MAX_GOVT_RETIREMENT
```

بالاعتماد على مبادئ تحليل الحدود، يجب فحص ثلاث حالات:

الحالة	وصف الحالة
1	الحالة الأولى معرفة لهذا الشرط الصحيح من أجل m_employee[ID].governmentRetirementWithheld < MAX_GOVT_RETIREMENT هو الحالة الأولى على الجانب الصحيح من الحدود. ولهذا تعد حالة الاختبار الأولى m_employee[ID].governmentRetirementWithheld إلى MAX_GOVT_RETIREMENT – 1 تم توليد حالة الاختبار هذه للتو.
3	الحالة الثالثة معرفة، لهذا الشرط الخاطئ من أجل m_employee[ID].governmentRetirementWithheld < MAX_GOVT_RETIREMENT هو على الجانب الخاطئ من الحدود. لهذا، تعد حالة الاختبار الثالثة m_employee[ID].governmentRetirementWithheld إلى MAX_GOVT_RETIREMENT + 1 تم توليد حالة الاختبار هذه أيضًا للتو.
10	تم إضافة حالة الاختبار إضافية إلى الحالة مباشرةً على الحدود التي فيها: m_employee[ID].governmentRetirementWithheld = MAX_GOVT_RETIREMENT

حدود مركبة

كما ينطبق تحليل الحدود على القيم المسموح بها الأعظمية والأصغرية. في هذا المثال، من الممكن أن يكون أصغري أو أعظمي grossPay، أو companyRetirement، أو PersonalRetirementContribution، ولكن بما أن حساب هذه القيم هو خارج نطاق الإجرائية، لم تتم مناقشة حالات الاختبار الخاصة بهم أكثر هنا.

يحدث نوع أكثر دهاءً من شرط الحدود عندما تتضمن الحدود مزيج من المتغيرات. على سبيل المثال، إذا كان هناك متغيرين مضروبين ببعضهما، ماذا يحدث عندما كل منهما هي أعداد موجبة كبيرة؟ أو عندما أعداد سالبة كبيرة؟ أو 0؟ ماذا لو كل السلاسل النصية الممررة إلى إجرائية طويلة بشكل غير شائع؟

في المثال الحالي، من الممكن أن ترغب لرؤية ماذا يحدث للمتغيرات totalWithholdings، وtotalGovernmentRetirement، وtotalGovernmentRetirement، وtotalRetirement، عندما كل عنصر من مجموعة كبيرة من العمال لديه راتب كبير- لنقول مثلًا، مجموعة من المبرمجين براتب 250,000 دولار لكل واحد. (نستطيع دائمًا أن نتأمل) يستدعي هذا حالة اختبار أخرى:

الحالة	وصف الحالة
11	مجموعة كبيرة من العمل، لكل منهم راتب ضخم (يعتمد ما يشكل "كبير" على النظام المحدد الذي يتم تطويره)- على سبيل المثال، سنقول أنه 1000 عامل، راتب كل واحد منهم 250,000 دولار، ولا يتم لأي أحد منهم خصم ضرائب الضمان الاجتماعي وكلهم يرغبون بالخصم من أجل التقاعد.

إن حالة الاختبار بنفس السياق، ولكن بالمقابل سيكون هناك مجموعة صغيرة من الموظفين، كل منهم لديه راتب قدره 0.00 دولار:

الحالة	وصف الحالة
12	مجموعة من 10 عمال، راتب كل واحد منهم 0.00 دولار.

صفوف المعطيات السيئة

بصرف النظر عن التخمين، تظهر الأخطاء حول شروط الحدود، يمكنك تخمين واختبار عدة صفوف أخرى من البيانات السيئة. تتضمن صفوف اختبار المعطيات السيئة ما يلي

- معطيات قليلة جدًا (أو لا معطيات)
- معطيات كثيرة جدًا
- النوع الخاطئ من المعطيات (معطيات غير صالحة)
- الحجم الخاطئ للمعطيات
- معطيات غير مُهيأة

بعض حالات الاختبار قد تفكر فيها إذا اتبعت هذه الاقتراحات التي تمت تغطيتها للتو. على سبيل المثال، تم تغطية "المعطيات القليلة جدًا" بحالات الاختبار 2 و12، ومن الصعب التوصل إلى أي شيء من أجل "معطيات بحجم خاطئ". ومع ذلك، فإن صفوف المعطيات السيئة تؤدي إلى بعض الحالات الأخرى:

الحالة	وصف الحالة
13	مصفوفة من 100,000,000 عامل. اختبارات للمعطيات الكثيرة جدًا. بالتأكيد، كم ستختلف الكثيرة هذه من نظام إلى النظام ، ولكن على سبيل المثال، لنفترض أن هذا كثير جدًا.
14	راتب سالب. النوع الخاطئ من المعطيات.
15	عدد سلبي للعمال. النوع الخاطئ من المعطيات.

صفوف المعطيات الجيدة

عندما تحاول إيجاد أخطاء في برنامج، من السهل إغفال حقيقة أنه قد تحتوي الحالة الاسمية على خطأ. عادةً تُمثل الحالات الإسمية الموصوفة في قسم اختبار الأساس نوع واحد من المعطيات الجيدة. فيما يلي الأنواع الأخرى من المعطيات الجيدة التي تستحق التدقيق. يمكن أن يكشف التحقق من كل من هذه الأنواع عن أخطاء، وذلك بالاعتماد على الخطأ الذي يتم اختباره.

- الحالات الإسمية- وسط الطريق، والقيم المتوقعة
- الإعداد الطبيعي الأصغري
- الإعداد الطبيعي الأعظمي
- التوافق مع البيانات القديمة

الإعداد الطبيعي الأصغري مُفيد للاختبار ليس فقط لعنصر واحد، بل لمجموعة من العناصر. إنه مُشابه لشرط الحدود للعديد من القيم الصُغرى، ولكنه يختلف بأن يُنشأ مجموعة من القيم الصغرى خارج عن مجموعة المتوقعة بشكل طبيعي. أحد الأمثلة على ذلك هو حفظ جدول بيانات فارغ عند اختبار جدول. لاختبار معالج نصوص سيكون حفظ مستند فارغ. في حالة المثال الجاري، سيضيف اختبار الإعداد الطبيعي الأصغري حالة الاختبار التالي :

الحالة	وصف الحالة
16	مجموعة من عامل واحد. لاختبار الإعداد الطبيعي الأصغري

الإعداد الطبيعي الأعظمي هو عكس الأصغري. إنه مُشابه لاختبار الحدود، ولكن مرة ثانية، إنه يُنشأ مجموعة من القيم العظمى خارج مجموعة القيم المتوقعة. مثال على ذلك هو حفظ جدول بيانات بحجم "الحد الأقصى لحجم جداول البيانات" المعلن عنه في تغليف المنتج. أو طباعة جدول البيانات بحجم الحد الأقصى. بالنسبة إلى معالج النصوص، فإنه سيتم حفظ مستند من أكبر حجم موصى به. في حالة المثال الجاري، يعتمد اختبار

إعداد الطبيعي الأعظمي على عدد الأعظمي الطبيعي للعمال. افترض أن عددهم الطبيعي الأعظمي هو 500، عندها ستضيف حالة الاختبار التالية:

الحالة	وصف الحالة
17	مجموعة من 500 عامل. لاختبار الإعداد الطبيعي الأعظمي

يدخل النوع الأخير من اختبار المعطيات الطبيعية- اختبار التوافق مع المعطيات القديمة- عندما البرنامج أو الإجرائية هو استبدال لبرنامج قديم أو إجرائية قديمة. يجب على الإجرائية الجديدة أن تنتج نفس النتائج من أجل المعطيات القديمة، كما تفعل الإجرائية القديمة، إلا في الحالات التي كانت فيها الإجرائية القديمة معيبة. هذا النوع من الاستمرارية بين نسخ برنامج أو إجرائية هو أساس اختبار التراجع (`regression testing`)، الذي الغرض منه هو ضمان أن التصحيحات والتحسينات تحافظ على مستويات الجودة السابقة دون التراجع. في حالة المثال الجاري، لن يضيف معيار التوافق أية حالات اختبار.

استخدم حالات الاختبار التي يكون التعامل معها باليد سهلا

افترض أنك تكتب حالة اختبار لراتب شكلي؛ أنت تحتاج إلى راتب لا على التعيين، والطريقة التي تحصل فيها عليه هي أن تكتب "أعدادا عشوائية" أياً كانت الأعداد التي تكتبها يديك. سأجرب هذا:

1239078382346

حسنًا. هذا راتب كبير، أكثر بقليل من تريليون دولار، في الحقيقة، ولكن إذا قمت بقصه بحيث يكون واقعي إلى حد ما، سأحصل على 90,783.82 دولار.

الآن، افترض أن حالة الاختبار ناجحة- أي أنها وجدت خطأ. كيف تعرف أنها وجدت خطأ؟ حسنًا، من المفترض أنك تعرف ما هي الإجابة، وماذا يجب أن تكون لأنك حسبت الجواب الصحيح باليد. عندما تحاول القيام لحسابات باليد مع أرقام بشعة مثل 90,783.82 دولار، على كل حال، من المحتمل أن ترتكب خطأ في حسابات اليد، كما ستكتشف خطأ في برنامجك. وبالمقابل، رقم جميل مثل 20,000 دولار يجعل التعامل معه أسهل. من السهل إدخال الأرقام 0 في الألة الحاسبة، والضرب بالعدد 2 هو أسهل ما يمكن للمبرمجين بالقيام باستخدام أصابعهم.

من الممكن أن تفكر أن عدد بشع مثل 90,783.82 دولار سيكون أكثر عرضة للكشف عن الأخطاء، ولكنه ليس أكثر احتمالًا من أي رقم آخر في صف المكافئ الخاص به.

4.22 الأخطاء النموذجية

هذا القسم مُخصص لاقتراح أنك تستطيع اختبار بشكل أفضل عندما تعرف أكثر ما يمكن عن عدوك: الأخطاء.

ماهي الصفوف التي تحوي على معظم الأخطاء؟



أنه من الطبيعي أن نفترض أن العيوب موزعة بالتساوي في جميع أنحاء الشفرة الخاصة بك. إذا كان لديك بمعدل 10 عيوب لكل 1000 سطر من الشفرة، من الممكن أن تفترض أنه سيكون لديك عيب واحد في الصف الذي يحتوي على 100 سطر من الشفرة. هذا افتراض طبيعي، لكنه خاطئ.

كتب كابرز جونز تقرير أن برنامج تحسين الجودة المركزة في IBM حدد 31 صف مُعرضة للخطأ من أصل 425 صف في نظام IMS. ثم تمت إعادة إصلاح هذه الصفوف (31) أو تمت إعادة تطويرها، وفي أقل من سنة، تم تخفيض العيوب التي أبلغ عنها المستهلك ضد نظام IMS من عشرة إلى واحد. خُفضت تكاليف الصيانة الكلية بحوالي 45 بالمئة. تحسن رضا العملاء من "غير مقبول" إلى "جيد" (جونز 2000).

تميل معظم الأخطاء إلى التموضع في عدد قليل من الإجراءات المعيبة للغاية. فيما يلي العلاقة العامة بين الأخطاء والشفرة:

- تم العثور على ثمانين بالمائة من الأخطاء في 20 بالمائة من صــفوف أو إجرائيات مشــروع (أندرز 1975، غريميليون 1984، بويم 1987، شول وآخرون 2002).
- تم العثور على خمســين بالمائة من الأخطاء في 5 بالمائة من صــفوف مشــروع (جونز 2000).

قد لا تبدو هذه العلاقات مهمة حتى تتعرف على بعض النتائج الطبيعية. أولًا، تُساهم 20 بالمئة من إجرائيات المشروع بـ 80 بالمئة من كلفة التطوير (بويم 1987). هذا لا يعني بالضرورة أن نسبة 20 بالمئة التي تتكلف أكثر من غيرها هي نفس نسبة 20 بالمئة التي تحتوي على أكبر عدد من العيوب، ولكنها توحي إلى حد كبير بهذا.

ثانيًا، بغض النظر عن النسبة الدقيقة للتكلفة المُسببة من قبل الإجرائيات المعيبة بشكل أكبر، الإجرائيات المُعيبة بشكل كبير هي مُكلفة بشكل كبير. في دراسة كلاسيكية في الأعوام 1960، أنجزت شركة IBM تحليل لأنظمة التشغيل OS/360 لديها، ووجدت أن الأخطاء لم تكن موزعة بشكل متساوي في كل الإجرائيات، ولكنها كانت مُركزة في بعضها. تم إيجاد أن هذه الإجرائيات المُعرضة للأخطاء هي " أغلى الكيانات في البرمجة "(جونز 1986a). احتوت على ما يصل إلى 50 من العيوب لكل 1000 سطر من الشفرة،

وكثيرًا ما كان إصلاحها يُكلف 10 أضعاف ما يتطلبه تطوير النظام بأكمله. (وشملت التكاليف دعم العملاء والصيانة في الموقع).

ثالثًا، الآثار المترتبة للإجرائيات المكلفة من أجل التطوير واضحة.[1] كما يقول التعبير القديم "الوقت هو المال". النتيجة الطبيعية هي أن "الوقت هو المال"، وإذا كنت تستطيع قطع ما يقرب من 80 في المئة من التكلفة عن طريق تجنب الإجرائيات المزعجة، يمكنك خفض كمية كبيرة من الجدول الزمني أيضًا. هذا توضيح للمبدأ العام في جودة البرمجيات: تحسين الجودة يُحسن الجدول الزمني للتطوير ويقلل من تكاليف التطوير.

رابعًا، الآثار المترتبة على تجنب الإجرائيات المزعجة للصيانة هي واضحة على حد السواء. يجب أن تُركز نشاطات الصيانة على تحديد، وإعادة تصميم، وإعادة الكتابة من الألف إلى الياء تلك الإجراءات التي تم تحديدها على أنها معرضة للأخطاء. في مشروع IMS المذكور سابقًا، ارتفعت إنتاجية IMS بنحو 15 في المائة بعد استبدال الصفوف المعرضة للأخطاء (جونز 2000).

الأخطاء حسب التصنيف

حاول العديد من الباحثين تصنيف الأخطاء حسب النوع وتحديد مدى حدوث كل نوع من الأخطاء[2]. لدى كل مُبرمج قائمة الأخطاء التي كانت مزعجة بشكل خاص: أخطاء الخطوة الواحدة، نسيان إعادة تهيئة متغير حلقة، وإلى آخره. تقدم قوائم التحقق المقدمة في جميع أنحاء الكتاب المزيد من التفاصيل.

جمّع بوريس بيزر بيانات من عدة دراسات، ووصل إلى تصنيف استثنائي مُفصل للأخطاء. فيما يلي مُلخص لنتائجه:

الهيكل	25.18%
المعطيات	22.44%
الوظائف كما نُفذت	16.19%
البناء	9.88%
التكامل	8.98%
متطلبات الوظائف	8.12%
تعريف الاختبار أو التنفيذ	2.76%
النظام، معمارية البرمجيات	1.74%
غير مُحدد	4.71%

[1] **إشارة مرجعية**: صف آخر من الإجرائيات الذي يميل لأن يحتوي على العديد من الاخطاء هو صف الإجرائيات المُعقدة جدًا. لمزيد من التفاصيل عن تحديد وتبسيط الإجرائيات، انظر "إرشادات توجيهية عامة لتخفيض التعقيد" في القسم 19.6.

[2] **إشارة مرجعية:** للحصول على قائمة بكل قوائم التحكم في هذا الكتاب، انظر القائمة التالية لجدول محتويات هذا الكتاب.

أبلغ بيزر عن نتائجه لخانتين عشريتين محددتين، ولكن البحث في أنواع الأخطاء كان بالعموم غير حاسم. كتبت أبحاث مختلفة عن أنواع مختلفة من الأخطاء، والدراسات التي كتبت عن أنواع مُشابهة من الأخطاء، وصلت إلى نتائج مُختلفة بشكل كبير، تلك النتائج تختلف بنسبة 50 بالمئة بدلًا من مائة من نقطة مئوية.

مع الاختلافات الكبيرة في نتائج الدراسات، على الأرجح لم ينتج تجميع النتائج من عدة دراسات كما فعل بيزر أية بيانات ذات معنى. ولكن حتى لو كانت البيانات غير حاسمة، فإن بعضها موحي. فيما يلي بعض الاقتراحات التي يمكن استخلاصها من هذه البيانات:

إن نطاق معظم الأخطاء محدود بشكل عادل. وجدت إحدى الدراسات أن 85 بالمئة من الأخطاء يمكن إصلاحها بدون تعديل أكثر من إجرائية واحدة (إندريس 1975).

العديد من الأخطاء خارج مجال البناء. بإجراء سلسلة من 97 مقابلة، وجد الباحثون أن أكثر ثلاث مصادر شائعة للأخطاء هي: المعرفة القليلة في مجال التطبيق، والمتطلبات المتعارضة والمتأرجحة، وانهيار الاتصال والتنسيق (كورتيس وكراسنر واسكو 1988).

معظم أخطاء البناء هي بسبب خطأ المبرمج.[1] وجد زوج من الدراسات أُجريت منذ عدة سنوات أن من المجموع الكلي للأخطاء، تحدث تقريبا 95 بالمئة من الأخطاء من قبل مبرمج، و2 بالمئة من قبل برمجيات النظام (المُترجم المبرمجي ونظام التشغيل)، و2 بالمئة من قبل بعض البرمجيات الأخرى، و1 بالمئة من قبل العتاد الصلب (براون وسامسون 1973، وأوستراند وويكر 1984). تُستخدم برمجيات الأنظمة وأدوات التطوير من قبل العديد من الناس اليوم، أكثر مما كان في الأعوام 1970 و1980، ولهذا تخميني الأفضل هو أنه، اليوم، النسبة الأعلى من الأخطاء هي خطأ المبرمجين.

تعتبر الأخطاء الكتابية مصدرًا شائعًا للمشكلات. وجدت إحدى الدراسات أن نسبة 36 بالمئة من أخطاء البناء هي أخطاء كتابية (فايس 1975). ووجدت دراسة عام 1987 لما يقارب من ثلاث مليون سطر برمجي من برنامج ديناميكيات الطيران، أن نسبة 18 بالمئة من كل الأخطاء كانت أخطاء كتابية (كارد 1987). وجدت دراسة أخرى أن 4 بالمئة من كل الأخطاء كانت أخطاء إملائية في الرسائل (إندريس 1975). في أحد برامجي، وجد زميل لي عدة أخطاء إملائية عن طريق تشغيل جميع السلاسل النصية من الملف القابل للتنفيذ من خلال المدقق الإملائي. ويدخل بالحسبان الاهتمام بالتفاصيل. إذا كنت تشك بهذا، خذ

[1] إذا كنت ترى أثار حافر، فكر بالأحصنة- وليس بالحمير الوحشية. من الممكن ألا يكون نظام التشغيل مُعطل. ومن الممكن أن تكون قاعدة البيانات فقط على ما يرام- اندي هانت وديف توماس.

بعين الاعتبار أن ثلاثة من أغلى أخطاء البرمجيات في كل العصور- كلفت 1.6 بليون دولار، و900 مليون دولار، و245 مليون دولار- احتوت على تغيير محرف واحد في برنامج صحيح سابقًا (واينبرغ 1983).

سوء فهم التصميم هو موضوع متكرر في دراسات أخطاء المبرمج. وجدت الدراسة التجميعية لبيزر، أن 16 بالمئة من الأخطاء نمت من سوء التفسير للتصميم (لبيزر 1990). وجدت دراسة أخرى أن 19 بالمئة من الأخطاء هي نتيجة سوء الفهم للتصميم (فيس 1975). من المفيد أن تأخذ الوقت الذي تحتاجه لفهم التصميم جيدًا. مثل هكذا وقت لا يُنتج أرباح فورية – فقد يبدو أنك لا تعمل - ولكن هذا يؤتي ثماره على مدى عمر المشروع.

معظم الأخطاء سهلة الإصلاح. يمكن إصلاح 85 بالمئة من الأخطاء بأقل من بضع ساعات. وحوالي 15 بالمئة من الأخطاء يمكن إصلاحها في بضع ساعات إلى بضعة أيام. وحوالي 1 بالمئة من الأخطاء تأخذ وقت أطول (فايس 1975، أوستراند وويكر 1984، جرادي 1992). هذه النتيجة تدعمها ملاحظة باري بويهم بأن حوالي 20 بالمئة من الأخطاء تأخذ حوالي 80 بالمئة من الموارد لإصلاحها (بويهم 1987). تجنّب الكثير من الأخطاء الصعبة بالقيام بمراجعات المتطلبات والتصميم الأصلية. تعامل مع الأخطاء الصغيرة العديدة بقدر ما تستطيع.

إنها فكرة جيدة قياس تجارب مؤسستك الخاصة مع الأخطاء. تُشير النتائج المتنوعة الموضوعة في هذا القسم بأنه لدى الناس في منظمات مختلفة تجارب مختلفة جدًا. هذا يجعل من الصعب تطبيق تجارب منظمات أخرى على منظمتك. تُعارض بعض النتائج الحدس العام؛ من الممكن أنك تحتاج لتزويد حدسك بأدوات جديدة. خطوة أولى جيدة هي بالبدء بقياس عملية التطوير الخاصة بك حتى تعرف أين توجد المشاكل.

نسبة الأخطاء الناتجة عن البناء الخاطئ

إذا كانت البيانات التي تُصنف الأخطاء غير حاسمة، كذلك الكثير من البيانات تُعزي سبب الأخطاء إلى أنشطة التطوير المختلفة. شيء مؤكد هو أن البناء يؤدي دائمًا إلى عدد كبير من الأخطاء. في بعض الأحيان، يُجادل بعض الأشخاص بأن الأخطاء المُسببة من قبل عملية البناء هي أخطاء رخيصة الإصلاح، أكثر من الأخطاء المُسببة من قبل المتطلبات أو التصميم. من الممكن أن يكون إصلاح أخطاء البناء المستقلة أرخص، لكن الدليل لا يدعم هكذا ادعاء حول التكلفة الإجمالية.

فيما يلي استنتاجاتي:

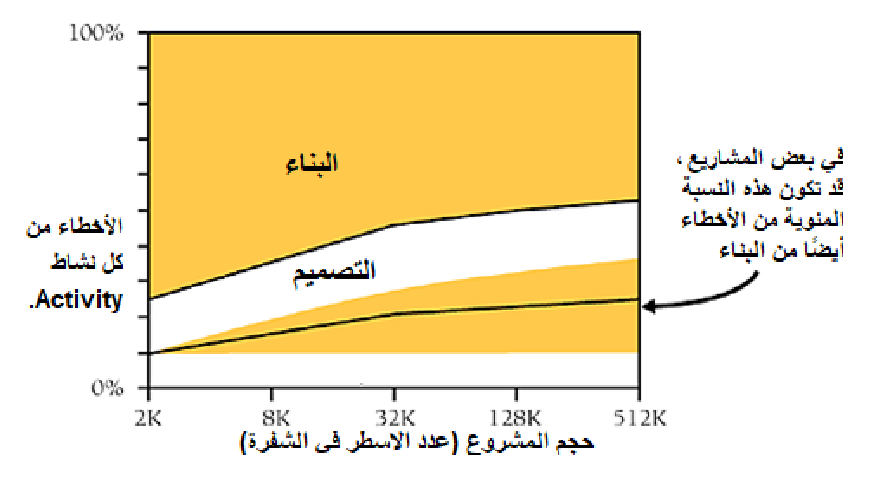


- في المشاريع الصغيرة، تُشكل عيوب البناء الجزء الأكبر من الأخطاء. في إحدى الدراسات عن كتابة شفرة الأخطاء في مشروع صغير (1000 سطر من الشفرة)، نتجت 75 بالمئة من الأخطاء بسبب

عملية كتابة الشفرة، بالمقارنة مع 10 بالمئة من الأخطاء بسبب المتطلبات، و15 بالمئة بسبب التصميم (جونز 1986). يظهر انهيار الخطأ هذا ليُمثل الكثير من المشاريع الصغيرة.

- تُشـــكّل عيوب البناء 35 بالمئة على الأقل من مجموع العيوب بغض النظر عن حجم المشـــروع. على الرغم من أن نسـبة عيوب البناء أصـغر في المشـاريع الكبيرة، إلا أنها لا تزال تُمثل 35 بالمئة على الأقل من جميع العيوب (بيريز 1990، جونز 2000). كتب بعض الباحثين عن نســب في مجال 75 بالمئة، حتى في المشـاريع الضـخمة (جرادي 1987). بشـكل عام، كلما كان الفهم أفضـل لمجال التطبيق، كلما كان الهيكل العام أفضــل. ومن ثم تميل الأخطاء إلى التركُّز في عملية التصـــميم التفصـــيلي وعملية التشفير (باسيلي وبريكون 1984).
- لا تزال أخطاء عملية البناء باهظة الثمن، على الرغم من أنها أرخص في الإصـلاح من أخطاء المتطلبات وأخطاء التصميم. ووجدت دراسة لمشروعين ضخمين في شركة هيوليت باكارد (Hewlett-Packard) أن متوسـط تكلفة عيب عملية البناء يبلغ 25-50 بالمئة من قدر ما يصـلح من خطأ التصـميم المتوسـط (جرادي 1987). عندما تم اكتشـــاف العدد الأكبر من عيوب البناء في التســـوية الكلية، كانت التكلفة الإجمالية لإصلاح عيوب البناء ضعف أو أكثر بمقدار تكلفة عيوب التصميم.


يقدم الشكل 22-2 فكرة تقريبية عن العلاقة بين حجم المشروع ومصدر الأخطاء.

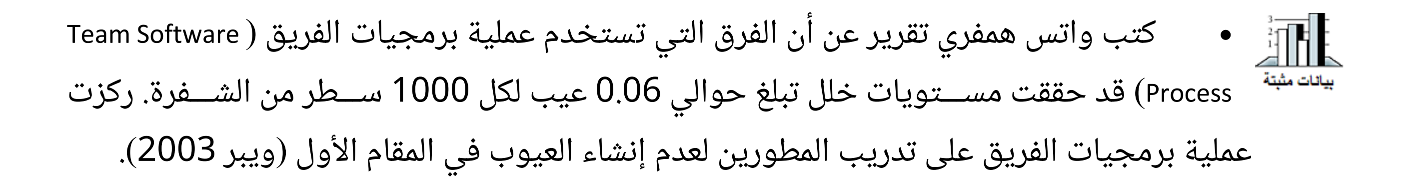


الشكل 22-2 كلما زاد حجم المشروع، كلما نقص عدد الأخطاء المرتكبة أثناء عملية البناء. ومع ذلك، تُشكل أخطاء عملية البناء حوالي 45-75 بالمئة من كل الأخطاء، حتى في المشاريع الأكبر.

كم عدد الأخطاء الذي تتوقع أن تكتشفها؟

يختلف عدد الأخطاء التي يجب أن تتوقع العثور عليها وفقًا لجودة عملية التطوير التي تستخدمها. فيما يلي مجال الاحتمالات:

- يبلغ متوسط خبرة الصناعة حوالي 1-25 خطأ لكل 1000 سطر من الشفرة للبرامج التي يتم تسليمها. تم تطوير البرمجيات عادةً باستخدام خليط من التقنيات (بوهم 1981، غريميليون 1984 ، يورون 1989a ، جونز 1998 ، جونز 2000 ، ويبر 2003). إن الحالات التي تحتوي على عُشر كمية الأخطاء هذه هي نادرة؛ أما الحالات التي لديها 10 مرات أكثر من الأخطاء تميل إلى أن تكون غير مُبلغ عنها. (ربما لا يتم الانتهاء منها).
- يواجه قسم التطبيقات في مايكروسوفت حوالي 10-20 عيوب لكل 1000 سطر من الشفرة أثناء الاختبار الداخلي و0.5 عيب لكل 1000 سطر من الشفرة في المنتج الصادر (مور 1992). إن التقنية المستخدمة لتحقيق هذا المستوى هي مزيج من تقنيات قراءة الشفرة الموضحة في القسم 4.21، "أنواع أخرى من ممارسات التطوير التعاوني" والاختبار المستقل.
- طورت شركة هارلان ميلز تقنية "تطوير غرف الأبحاث"، وهي تقنية تمكنت من تحقيق معدلات منخفضة تصل إلى 3 عيوب لكل 1000 سطر من الشفرة أثناء الاختبار الداخلي و0.1 عيب لكل 1000 سطر من الشفرة في المنتج المصدر (كوب وميلز 1990). حققت بعض المشاريع - على سبيل المثال، برنامج المكوك الفضائي - مستوى من 0عيب في 500000 سطر من الشفرة باستخدام نظام من مناهج التطوير الرسمية، ومراجعات الأقران، والاختبارات الإحصائية (فيشمان 1996).



- كتب واتس همفري تقرير عن أن الفرق التي تستخدم عملية برمجيات الفريق (Team Software Process) قد حققت مستويات خلل تبلغ حوالي 0.06 عيب لكل 1000 سطر من الشفرة. ركزت عملية برمجيات الفريق على تدريب المطورين لعدم إنشاء العيوب في المقام الأول (ويبر 2003).

تؤكد نتائج عملية برمجيات الفريق ومشروعات غرف الأبحاث على نسخة أخرى من المبدأ العام لجودة البرمجيات: من الأرخص بناء برامج عالية الجودة بدلاً من بناء برامج منخفضة الجودة وإصلاحها. كانت الإنتاجية لمشروع غرفة الأبحاث كامل حجمه 80000 خط هو 740 سطر من الشفرة لكل شهر عمل. إن متوسط معدل الصناعة لشفرة مفحوصة بالكامل هو أقرب إلى 250-300 سطر لكل شهر عمل، بما في ذلك جميع النفقات غير المرتبطة بكتابة الشفرة (كوسومانو وآخرون 2003). تأتي توفيرات التكلفة والإنتاجية من حقيقة أنه لا يتم تخصيص أي وقت تقريبًا لتصحيح الأخطاء في "عملية برمجيات الفريق" أو مشروعات غرف الأبحاث. لا يتم صرف وقت في تصحيح الأخطاء؟ إنه فعلا يستحق!

أخطاء في الاختبار نفسه



ربما تكون لديك تجربة مثل هذه: تم اكتشاف وجود خطأ في البرنامج. لديك بعض المشاعر والتخمينات المباشرة حول جزء ما من الشفرة قد يكون خاطئًا، ولكن يبدو أن كل الشفرة صحيحة.

يمكنك تشغيل عدة حالات اختبار أخرى لمحاولة تنقيح الخطأ، ولكن جميع حالات الاختبار الجديدة تعطي نتائج صحيحة. تقضي عدة ساعات في القراءة وإعادة قراءة الشفرة وحساب النتائج يدويًا. هذه النتائج تتحقق من كل شيء. وبعد بضع ساعات أخرى، هناك شيء ما يجعلك تعيد فحص معطيات الاختبار. وجدتها! الخطأ في معطيات الاختبار! حيث تشعر بأنك أحمق لتضييع ساعات من ملاحقة خطأ في معطيات الاختبار بدلًا من ملاحقته في الشفرة!

بيانات مثبتة

هذه تجربة شائعة. غالبًا ما تحتوي أو من المحتمل أن تحتوي حالات الاختبار على أخطاء أكثر من الشفرة التي يجري اختبارها (فييلاند 1983، جونز 1986، جونسون 1994). من السهل العثور على الأسباب- خاصة عندما يكتب المطور حالات الاختبار. تميل حالات الاختبار إلى أن يتم إنشاؤها بسرعة، بدلًا من تصميمها بنائها بعناية. وغالبا ما ينظر إليها على أنها اختبارات لمرة واحدة ويتم تطويرها بعناية تتناسب مع تطوير شيء سيتم التخلص منه.

يمكنك القيام بالعديد من الأشياء لإنقاص عدد الأخطاء في حالات الاختبار لديك:

تحقق من عملك. طوّر حالات الاختبار بالعناية نفسها التي تطور فيها الشفرة. تتضمن هكذا عناية بشكل أساسي فحص مُضاعف لاختبارك. افحص كل سطر من شفرة الاختبار، كما لو كنت تفحص شفرة الانتاج. تعد عمليات فحص وتفتيش معطيات الاختبار مناسبة.

خطط لحالات الاختبار كما تطور برمجيتك الخاصة. يجب أن يبدأ التخطيط الفعال لاختبارك في مرحلة المتطلبات أو بمجرد حصولك على مهمة تطوير البرنامج. يُساعدك هذا على تجنُب حالات الاختبار المستندة على افتراضات خاطئة.

احفظ حالات الاختبار لديك. اصرف وقت قليل الجودة مع حالات الاختبار لديك. ومن ثم احفظهم من أجل اختبار التراجع ومن أجل العمل على الإصدار الثاني. من السهل تبرير المشكلة إذا كنت تعرف أنك ستحتفظ بهم بدلًا من التخلص منها.

ضع اختبارات الوحدة في هيكل اختبار. اكتب التعليمات البرمجية لاختبارات الوحدة أولًا، ولكن ادمجهم في منظومة هيكل اختبار (مثل `JUnit`)، حتى تكمل كل اختبار منهم. يمنع وجود هيكل اختبار مُدمج الميل، المذكور للتو، للتخلص من حالات الاختبار.

5.22 أدوات دعم الاختبار

يُلخص هذا القسم أنواع الاختبار التي يمكن أن تشتريها تجاريًا أو تبنيها بنفسك. لن يقوم هذا القسم بتسمية منتجات محددة لأنه من السهل أن تفقد صلاحيتها بحلول الوقت الذي تقرأ فيه هذا. اطّلع على مجلة المبرمج المُفضلة لديك من أجل معرفة منتجات الاختبار الأكثر حداثة.

بناء السقالات (Scaffolding) لاختبار الصفوف الفردية

يأتي مصطلح "سقالة" من عملية بناء مبنى. تُبنى السقالات ليتمكن العمال من الوصول إلى أجزاء من مبنى لا يمكنهم الوصول إليه بطريقة أخرى. تُبنى السقالات البرمجية لغرض وحيد هو تسهيل تنفيذ الشفرة.

أحد أنواع السقالات هو عبارة عن صف يتم تجنيده بحيث يمكن استخدامه من قبل صف آخر يتم اختباره.[1] يُطلق على هذا الصف اسم كائن وهمي" mock object" أو كائن زائف (كعب)[2] stub (ماكينون، فريمان، كريج 2000؛ توماس وهانت 2002). يمكن استخدام الأسلوب ذاته مع إجرائيات المستوى المنخفض، التي تُدعى "إجرائيات زائفة" stub routines. يمكنك تشكيل كائن وهمي أو إجرائية زائفة أكثر أو أقل واقعية، وذلك اعتمادًا على مدى الصدق الذي تحتاجه. في هذه الحالات، يمكن للسقالات أن

- تُعيد التحكم فورًا، بدون اتخاذ أي إجراء.
- تختبر البيانات التي تغذيها.
- تطبع رسالة تشخيص، من الممكن أن تكون صدى لوسطاء الدخل، أو تسجيل رسالة في ملف.
- تحصل على قيم الإرجاع من الدخل التفاعلي.
- تعيد جواب قياسي (موحد) بغض النظر على الدخل.
- حرق عدد دورات الساعة المخصصة للكائن الحقيقي أو الإجرائية.
- تعمل كنسخة بطيئة أو دهنية أو كبيرة أو أقل دقة من الكائن الحقيقي أو الإجرائية.

نوع آخر من السقّالة هو إجرائية مزيفة تستدعي الإجرائية الحقيقية الجاري اختبارها. هذا يسمى "سائق (driver)" أو، في بعض الأحيان، "أداة اختبار". يمكن لهذا السقّالة أن

- تستدعي كائن بمجموعة ثابتة من المدخلات.
- المطالبة الفورية بالحصول على الدخل بشكل تفاعلي واستدعاء الكائن معه.
- تأخذ المعاملات من سطر الأمر (في أنظمة التشغيل التي تدعم هذا) وتستدعي الكائن.

[1] **قراءة متعمقة**: للمزيد من الأمثلة الجيدة عن "السقالة"، انظر مقالة جون بنتلي "مسألة صغيرة من البرمجة" في درر البرمجة، الإصدار الثاني (2000)

[2] - mock object الكائن الوهمي وهو كائن يأخذ مكان كائن" حقيقي "بطريقة تجعل الاختبار أسهل وأكثر جدوى.

- تقرأ المعاملات من ملف وتستدعي الكائن.
- تشغيل المجموعات مسبقة التعريف من معطيات الدخل في استدعاءات عديدة للكائن.

النوع الأخير من السقالات هو الملف الوهمي،[1] وهو نسخة صغيرة من الشيء الحقيقي الذي يحتوي على نفس أنواع المكونات التي يحتوي عليها ملف كامل الحجم. يقدم ملف صغير وهمي اثنين من المزايا. نظرًا لأنه صغير، يمكنك معرفة محتوياته بدقة ويمكن أن يكون متأكدًا إلى حد معقول من أن الملف نفسه خالي من الأخطاء. ولأنك قمت بإنشائه خصيصًا للاختبار، يمكنك تصميم محتوياته بحيث يكون أي خطأ في استخدامه واضحًا.

من الواضح أن بناء السقالات يتطلب بعض العمل،[2] ولكن إذا تم اكتشاف خطأ في أي وقت في صف، يمكنك إعادة استخدام السقالات. وتوجد العديد من الأدوات لتسهيل إنشاء كائنات وهمية وغيرها من السقالات. إذا كنت تستخدم السقالات، يمكن أيضًا اختبار الصف دون التعرض لخطر تأثرها بالتفاعلات مع الصفوف الأخرى. تكون السقالات مفيدة بشكل خاص عند تضمين خوارزميات دقيقة. من السهل أن تتعثر في حالة تستغرق عدة دقائق لتنفيذ كل حالة اختبار نظرًا لأن الشفرة التي يتم تنفيذها مضمّنة في شفرة أخرى. تسمح لك السقالات بتنفيذ الشفرة بشكل مباشر. إن الدقائق القليلة التي تقضيها في بناء السقالات لتنفيذ الشفرة المدفونة بعمق يمكن أن توفر ساعات من وقت التصحيح.

يمكنك استخدام أي من أطر الاختبار العديدة المتاحة لتوفير السقالات لبرامجك (`JUnit`، و`CppUnit`، و`NUnit`، وما إلى ذلك). إذا كانت البيئة غير مدعومة من قِبل أحد أطر الاختبار الحالية، فيمكنك كتابة بضعة إجرائيات في صف، وتضمين إجرائية السقّالة `main()` في الملف لاختبار الصف، على الرغم من أن الإجرائيات المُختبرة غير مُعدة للوقوف لوحدها. يمكن للإجرائية الأساسية `main()` قراءة المعاملات من سطر الأوامر وتمريرها إلى الإجرائية المُختبرة، بحيث يمكنك تنفيذ الإجرائية من تلقاء نفسه قبل دمجه مع باقي البرنامج. عندما تقوم بدمج الشفرة، اترك الإجرائيات وشفرة السقالات التي تنفذها في الملف واستخدم الأوامر المسبقة أو التعليقات لإلغاء تنشيط شفرة السقالات. نظرًا لأنه يتم معالجتها مسبقًا، فإنها لا تؤثر على الشفرة القابلة للتنفيذ، ولأنها في أسفل الملف، فهي ليست مرئية. فلا ضرر من تركها. إنها هناك إذا احتجت إليها مرة أخرى، ولا تحرق الوقت الذي سيستغرقه إزالتها وأرشفتها.

أدوات المقارنة (Diff)

[1] **إشارة مرجعية:** إن الخط الفاصل بين أدوات الاختبار وأدوات تصحيح الأخطاء غامض. لمزيد من التفاصيل حول أدوات تصحيح الأخطاء، انظر القسم 23.5، "أدوات التصحيح-الواضحة وغير الواضحة بشكل كبير".

[2] `cc2e.com/2268`

اختبار التراجع، أو إعادة الاختبار[1]، هو أسهل بكثير إذا كان لديك أدوات آلية للتحقق من الخرج الفعلي مقابل الخرج المتوقع. إحدى الطرق السهلة للتحقق من الخرج المطبوع هي إعادة توجيه الخرج إلى ملف واستخدام أداة مقارنة الملفات مثل (`diff`) لمقارنة الخرج الجديد بالمخرجات المتوقعة التي تم إرسالها إلى ملف مسبقًا. إذا لم تكن قيم الخرج متشابهة، فقد اكتشفت خطأ تراجع(انحدار).

مولدات معطيات الاختبار

يمكنك أيضًا كتابة شفرة لتنفيذ أجزاء محددة من برنامج بشكل منهجي.[2] قبل بضع سنوات، طورت خوارزمية تشفير الملكية وكتبت برنامج تشفير الملفات لاستخدامه. كان الهدف من البرنامج هو تشفير ملف بحيث يمكن فك ترميزه فقط باستخدام كلمة المرور الصحيحة. لم يعمل التشفير فقط على تغيير الملف بشكل سطحي؛ بل غير المحتويات بأكملها. كان من الأهمية أن يكون البرنامج قادرًا على فك تشفير ملف بشكل صحيح، لأن الملف سيُدمر بطريقة أخرى.

أقوم بإعداد مولد معطيات اختبار ينفذ بالكامل أجزاء التشفير وفك التشفير الخاصة بالبرنامج. قام المولد بتوليد ملفات من أحرف عشوائية في أحجام عشوائية، من 0 كيلوبايت إلى 500 كيلوبايت. وولّد كلمات المرور من أحرف عشوائية في أطوال عشوائية من 1 إلى 255 حرف. بالنسبة لكل حالة عشوائية، ولّد المولد نسختين من الملف العشوائي، وتشفير نسخة واحدة، وإعادة تهيئة نفسه، وإلغاء تشفير النسخة، ثم مقارنة كل بايت في النسخة التي تم فك تشفيرها بالنسخة غير المُغيرة. إذا كان أي بايت مُختلف، قام المولد بطباعة كل المعلومات التي احتاجها لإعادة إظهار الخطأ.

لقد قيّمت حالات الاختبار إلى متوسط طول ملفاتي، 30 كيلوبايت، الذي كان أقصر بكثير من الحد الأقصى للطول وهو 500 كيلوبايت. إذا لم أقم بتقييم حالات الاختبار بطول أقصر، فستكون أطوال الملفات موزعة بشكل منتظم بين 0 كيلوبايت و500 كيلوبايت. كان متوسط طول الملف المُختبر 250 كيلوبايت. يعني متوسط الطول الأقصر أن بإمكاني اختبار المزيد من الملفات، وكلمات المرور، وشروط نهاية الملف وأطوال الملفات الفردية، والملابسات الأخرى التي قد تنتج أخطاء أكثر مما يمكن أن أحصل عليه مع أطوال عشوائية منتظمة.

كانت النتائج مرضية. بعد تشغيل حوالي 100 حالة اختبار فقط، وجدت خطأين في البرنامج. نشأت كلتا الحالتين من حالات خاصة لم تكن قد ظهرت قط في الممارسات العملية، لكنها كانت خاطئة مع ذلك، وكنت مسرورا بإيجادهم. بعد إصلاحها، قمت بتشغيل البرنامج لأسابيع، وقمت بتشفير وفك تشفير أكثر من 100000

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن اختبار التراجع، انظر "إعادة الاختبار (اختبار التراجع)" في القسم 22.6.

[2] cc2e.com/2275

ملف دون أية أخطاء. بالنظر إلى المجال من محتويات الملف، وأطواله، وكلمات المرور التي قمت باختبارها، يمكنني أن أؤكد بثقة أن البرنامج كان صحيحًا.

فيما يلي بعض الدروس من هذا:

- يمكن لمولدات المعطيات العشوائية المصممة بشكل صحيح توليد مجموعات غير معتادة من معطيات الاختبار التي لا تفكر فيها.
- يمكن لمولدات المعطيات العشوائية أن تُنفذ برنامجك بشكل أكثر دقة مما تستطيع أنت.
- يمكنك تحسين حالات الاختبار التي تم إنشاؤها عشوائيًا بمرور الوقت، بحيث يتم التركيز على مجال واقعي من الدخل. يركز هذا على الاختبار في المناطق التي يرجح أن يتم تنفيذها من قبل المستخدمين، مما يزيد من الموثوقية في تلك المناطق.
- تصميم الوحدات يُؤتي ثماره أثناء الاختبار. تمكنت من سحب شفرة التشفير وفك التشفير واستخدامها بشكل مستقل عن شفرة واجهة المستخدم، مما جعل مهمة كتابة برنامج اختبار مباشرة وواضحة.
- يمكنك إعادة استخدام برنامج اختبار إذا كان من الضروري تغيير الشفرة التي يختبرها هذا البرنامج. بمجرد أن صححت الأخطاء المبكرة، تمكنت من البدء في إعادة الاختبار على الفور.

مراقبات التغطية

بيانات مثبتة

يفيد كارل ويجيرز أن الاختبار الذي يتم إجراؤه بدون تغطية قياس الشفرة لا يُنفذ عادة سوى حوالي 50-60 بالمئة من الشفرة (ويجيرز 2002)[1]. أداة مراقبة التغطية هي أداة تتعقب الشفرة التي يتم تنفيذها، والشفرة التي لا يتم تنفيذها. تعتبر أداة مراقبة التغطية مفيدة بشكل خاص للاختبارات المنهجية، لأنها تخبرك ما إذا كانت مجموعة من حالات الاختبار تُنفذ الشفرة بشكل كامل. إذا قمت بتشغيل مجموعتك الكاملة من حالات الاختبار وتُشير أداة مراقبة التغطية إلى أنه لم يتم تنفيذ بعض من الشفرة، فإنك ستعلم أنك بحاجة إلى المزيد من الاختبارات.

مسجل المعطيات / التسجيل

يمكن لبعض الأدوات مراقبة البرنامج وجمع المعلومات حول حالة البرنامج في حالة حدوث فشل - على غرار "الصندوق الأسود" الذي تستخدمه الطائرات لتشخيص نتائج الأعطال. يساعد التسجيل القوي على تشخيص الخطأ، ويدعم خدمة فعالة بعد إصدار البرنامج.

[1] cc2e.com/2282

يمكنك إنشاء مسجل البيانات الخاص بك عن طريق تسجيل الأحداث الهامة في ملف. سجل حالة النظام قبل حدوث خطأ، وسجل تفاصيل عن شروط الخطأ الدقيقة. يمكن تجميع هذه الوظيفة في نسخة التطوير الخاص بالشفرة وتجميعها خارج النسخة المصدرة. بدلًا من ذلك، إذا قمت تنفيذ التسجيل باستخدام تخزين تشذيب ذاتي، وموقع مدروس ومحتوى رسائل الخطأ، يمكنك تضمين وظائف التسجيل في النسخ الصادرة.

المصححات الرمزية (Symbolic Debuggers)

المصحح الرمزي هو مكمل تكنولوجي لمسارات الشفرة وعمليات التفتيش.[1] لدى مصحح الأخطاء القدرة على التنقل خلال أسطر الشفرة سطر تلو سطر، وتتبع قيم المتغيرات، والتفسير الدائم للشفرة بالطريقة نفسها التي يعمل بها الحاسب. تعتبر عملية التنقل خلال جزء من الشفرة في مصحح الأخطاء ومراقبتها مفيدة بشكل كبير.

إن عملية تتبع الشفرة في مصحح الأخطاء في العديد من النواحي هي مشابهة لنفس عملية تتبع مبرمجين آخرين الشفرة خلال مراجعة. لا يوجد لدى زملائك المبرمجين أو لدى مصحح الأخطاء نفس النقاط العمياء التي تقوم بها أنت. الفائدة الإضافية من مصحح الأخطاء هي أنه أقل كثافة في العمل من مراجعة فريق من المبرمجين. تُعد مراقبة تنفيذ شفرتك ضمن مجموعة متنوعة من مجموعات بيانات الدخل بمثابة تأكيد جيد على أنك نفذت الشفرة التي كنت تنوي تنفيذها.

يُعد المصحح الجيد أداة جيدة للتعلم عن لغتك لأنك تستطيع أن ترى بالضبط كيفية تنفيذ التعليمات البرمجية في الشفرة. يمكنك التبديل بين طريقة عرض شفرة بلغة عالية المستوى وعرض طريقة المجمع (assembler) لمعرفة كيفية ترجمة الشفرة عالية المستوى إلى لغة المجمع. يمكنك مشاهدة التسجيلات والمكدس لمعرفة كيف يتم تمرير المُعاملات. يمكنك إلقاء نظرة على شفرتك الذي قام المجمع بتحسينها لمعرفة أنواع التحسينات التي يتم تنفيذها. لا علاقة لأي من هذه المزايا بالاستخدام المقصود للبرنامج المصحح - تشخيص الأخطاء التي تم اكتشافها بالفعل - ولكن الاستخدام المبتكر لمصحح الأخطاء ينتج عنه فائدة أبعد من ميزاته الاعتيادية.

[1] **إشارة مرجعية**: يختلف توفر المصححات وفقًا لنضج البيئة التقنية. لمزيد من التفاصيل عن هذه الظاهرة، انظر القسم 3.4، "موقعك على موجة التكنولوجيا".

اضطرابات النظام System Perturbers

تم تصميم فئة أخرى من أدوات دعم الاختبار لاضطراب النظام.[1] كثير من الناس لديهم قصص من البرامج التي تعمل 99 مرة من أصل 100 ولكن تفشل في المرة 100 مع نفس المعطيات. تكمن المشكلة تقريبًا بشكل دائم في فشل تهيئة متغير ما في مكان ما، ويكون من الصعب عادةً إعادة الإنتاج لأن 99 مرة من أصل 100 يحدث أن يكون المتغير غير المهيأ يساوي 0.

تشتمل أدوات دعم الاختبار في هذه الفئة من الأدوات على مجموعة متنوعة من الإمكانات:

- **ملء الذاكرة**. أنت تريد أن تتأكد من عدم وجود أي متغيرات غير مهيأة. تملأ بعض الأدوات الذاكرة بقيم عشوائية قبل تشغيل البرنامج بحيث لا يتم تعيين المتغيرات غير المهيأة بالقيمة 0 بدون قصد. في بعض الحالات، قد يتم تعيين الذاكرة إلى قيمة محددة. على سبيل المثال، في معالج x86، القيمة *0xCC* هي رمز لغة الجهاز من أجل نقطة توقف. إذا كنت تملأ الذاكرة باستخدام القيمة *0xCC* حصل لديك خطأ يؤدي إلى تنفيذ أمر لا ينبغي عليك فعله، فستضرب نقطة توقف في مصحح الأخطاء وتكتشف الخطأ.
- **اهتزاز الذاكرة**. في أنظمة تعدد المهام، يمكن لبعض الأدوات إعادة ترتيب الذاكرة أثناء تشغيل البرنامج بحيث يمكنك التأكد من عدم كتابة أية شفرة تعتمد على المعطيات في المواقع المطلقة بدلاً من المواقع النسبية.
- **فشل الذاكرة الانتقائية**. يمكن لمحرك الذاكرة أن يحاكي ظروف الذاكرة المنخفضة التي قد ينفد فيها البرنامج من الذاكرة، أو يفشل في طلب الذاكرة، أو يمنح عددًا عشوائيًا من طلبات الذاكرة قبل الفشل، أو يفشل في عدد عشوائي من الطلبات قبل منحه طلب. هذا مفيد بشكل خاص لاختبار البرامج المعقدة العاملة مع الذاكرة المخصصة بشكل ديناميكي.
- **التحقق من الوصول للذاكرة (التحقق من الحدود)**. يُراقب مدققون الحدود عمليات المؤشر للتأكد من أن المؤشرات تتصرف بأنفسهم. مثل هكذا أداة مفيدة للكشف عن المؤشرات غير المهيأة أو المُعلّقة.

قواعد بيانات الأخطاء

إحدى أدوات الاختبار القوية هي قاعدة بيانات الأخطاء التي تم الإبلاغ عنها.[2] تُعد قاعدة البيانات هذه أداة إدارية وتقنية. تسمح لك قاعدة البيانات بالتحقق من الأخطاء المتكررة، وتلاحق معدل اكتشاف الأخطاء الجديدة وتصحيحها، وتتبع حالة الأخطاء المفتوحة والمغلقة وشدتها. للحصول على تفاصيل حول المعلومات التي يجب عليك الاحتفاظ بها في قاعدة بيانات الأخطاء، راجع القسم 7.22، "الاحتفاظ بسجلات الاختبار".

[1] cc2e.com/2289
[2] cc2e.com/2296

6.22 تحسين اختبارك

إن خطوات تحسين الاختبار مشابهة لخطوات تحسين أية عملية أخرى. يجب أن تعرف بالضبط ما تفعله العملية بحيث يمكنك تغييرها قليلًا وملاحظة تأثيرات الاختلاف. عندما تُلاحظ تغييرًا له تأثير إيجابي، فأنت تقوم بتعديل العملية بحيث تصبح أفضل قليلًا. تصف الأقسام التالية كيفية القيام بذلك باستخدام الاختبار.

التخطيط للاختبار

إحدى مفاتيح الاختبار الفعّال هو التخطيط منذ بداية المشروع لاختباره.[1] إن وضع الاختبار على نفس المستوى من الأهمية مثل التصميم أو كتابة الشفرة يعني أنه سيتم تخصيص الوقت له، وسيتم اعتباره مهمًا، وسيكون كعملية عالية الجودة. يعد تخطيط الاختبار أيضًا عنصرًا في جعل عملية الاختبار قابلة للتكرار. إذا لم تتمكن من تكرار الاختبار، فلا يمكنك تحسينه.

إعادة الاختبار (اختبار التراجع)

لنفترض أنك اختبرت منتجًا بعناية ولم تعثر على أية أخطاء. ولنفترض أن المنتج قد تغير بعد ذلك في منطقة واحدة، وتريد أن تتأكد من أنه لا يزال يمر بجميع الاختبارات التي أجراها قبل التغيير – وأن هذا التغيير لم يقدم أي عيوب جديدة. يُسمى الاختبار المصمم للتأكد من أن البرنامج لم يتخذ خطوة إلى الوراء، أو أنه "متراجع"، باسم "اختبار التراجع".

من المستحيل تقريبًا إنتاج منتج برمجي عالي الجودة ما لم يكن بإمكانك إعادة اختباره بشكل منهجي بعد إجراء التغييرات. إذا قمت بإجراء اختبارات مختلفة بعد كل تغيير، فلا يوجد لديك طريقة للتأكد من عدم وجود عيوب جديدة. وبالتالي، يجب أن يقوم اختبار التراجع بإجراء الاختبارات نفسها في كل مرة. في بعض الأحيان يتم إضافة اختبارات جديدة مع نضج المنتج، ولكن يتم الاحتفاظ بالاختبارات القديمة أيضا.

الاختبار الآلي



الطريقة العملية الوحيدة لإدارة اختبار التراجع هي جعله اختبار آلي. يصبح الناس خَدرين من إجراء الاختبارات نفسها عدة مرات ومن رؤية نتائج الاختبار نفسها عدة مرات. حيث يصبح من السهل للغاية تجاهل الأخطاء، التي تُفشل الغرض من اختبار التراجع. وجد خبير الاختبار بوريز بيزر أن معدل الخطأ

[1] إشارة مرجعية: جزء من التخطيط للاختبار هو إضفاء الطابع الرسمي على خططك في الكتابة. للحصول على مزيد من المعلومات حول توثيق الاختبار، راجع قسم "الموارد الإضافية" في نهاية الفصل 32.

في الاختبار اليدوي يمكن مقارنته بمعدل الخطأ في الشفرة التي يجري اختبارها. ويخمن أنه في الاختبار اليدوي، يتم تنفيذ حوالي فقط نصف جميع الاختبارات بشكل صحيح (جونسون 1994).

فيما يلي فوائد الاختبار الآلي:

- لدى الاختبار التلقائي فرصة أقل من الاختبار اليدوي ليكون خاطئًا.
- بعد إجراء الاختبار تلقائيًا، سيكون متاحًا لبقية المشروع مع جهد قليل من جانبك.
- إذا كانت الاختبارات آلية، يمكن تشغيلها بشكل متكرر لمعرفة ما إذا كانت أية تسجيلات للشفرة قد كسرت الشفرة أم لا. يمثل أتمتة الاختبار جزءًا من أساس ممارسات الاختبار المكثفة، مثل الاختبار اليومي للبناء والدخان والبرمجة المتطرفة.
- تُحسن الاختبارات الآلية من فرص اكتشاف أي مشكلة معينة في أقرب وقت ممكن، الأمر الذي يميل إلى تقليل العمل المطلوب لتشخيص المشكلة وتصحيحها.
- توفر الاختبارات الآلية شبكة أمان لتغييرات الشفرة واسعة النطاق، لأنها تزيد من فرص الاكتشاف السريع للعيوب التي تم إدخالها أثناء التعديلات.
- تعتبر الاختبارات الآلية مفيدة بشكل خاص في بيئات التكنولوجيا الجديدة المتقلبة لأنها تطرد التغييرات في البيئات عاجلاً وليس آجلًا.[1]

توفر الأدوات الرئيسية المستخدمة لدعم الاختبار الآلي سقالات اختبار، وتولّد الدخل، وتلتقط الخرج، وتقارن الخرج الفعلي مع الخرج المتوقع. ستنفذ المجموعة المتنوعة من الأدوات المُناقشة في القسم السابق بعض أو كل هذه الوظائف.

7.22 الاحتفاظ بسجلات الاختبار



بصرف النظر عن جعل عملية الاختبار قابلة للتكرار، فأنت بحاجة لقياس المشروع بحيث يمكنك معرفة ما إذا كانت التغييرات ستحسن المشروع أو تنزله مرتبه. فيما يلي مجموعة قليلة من المعطيات التي يمكنك جمعها لقياس مشروعك:

- الوصف الإداري للخلل (التاريخ المبلغ عنه، الشخص الذي قام بالإبلاغ عنه، العنوان أو الوصف، رقم البناء، الوقت المُثبت)
- الوصف الكامل للمشكلة
- الخطوات الواجب اتخذها للإبلاغ عن المشكلة

[1] **إشارة مرجعية**: لمزيد من التفاصيل حول العلاقات بين نضج التكنولوجيا وممارسات التطوير، انظر القسم 3.4، "موقعك في موجة التكنولوجيا".

- الحل المقترح لهذه المشكلة
- العيوب ذات الصلة
- خطورة المشكلة - على سبيل المثال، قاتلة، أو مزعجة، أو تجميلية
- أصل العيب: المتطلبات أو التصميم أو كتابة الشفرة أو الاختبار
- تصنيف فرعي لعيب كتالة الشفرة: خطأ الخطوة الواحدة، الإسناد غير الصحيح، فهرس مصفوفة غير صحيح، نداء سيء لإجرائية، وما إلى ذلك
- الصفوف والإجرائيات المُتغيرة عن طريق عملية الإصلاح
- عدد أسطر التعليمات البرمجية في الشفرة المتأثرة بالعيب
- الساعات المُستغرقة في إيجاد العيب
- الساعات المُستغرقة في إصلاح العيب

بمجرد أن تقوم بتجميع البيانات، يمكنك حساب عدد قليل من الأرقام لتحديد ما إذا كان مشروعك يزداد سوءًا أو يزداد صحةً:

- عدد العيوب الموجودة في كل صف، مرتبة من أسوأ درجة إلى الأفضل، وربما تكون مسواة حسب حجم الصف
- عدد العيوب الموجودة في كل إجرائية، مرتبة من أسوأ درجة إلى الأفضل، وربما تكون مسواة حسب حجم الإجرائية
- متوسط عدد ساعات الاختبار لكل عيب تم العثور عليه
- متوسط عدد العيوب الموجودة في كل صف
- متوسط عدد ساعات البرمجة لإصلاح كل عيب
- النسبة المئوية للشفرة المغطاة بحالات الاختبار
- عدد العيوب البارزة في كل تصنيف خطير

سجلات الاختبار الشخصية

بالإضافة إلى سجلات الاختبار على مستوى المشروع، قد تجد أنه من المفيد تتبع سجلات الاختبار الشخصية الخاصة بك. يمكن أن تتضمن هذه السجلات كلاً من قائمة التحقق من الأخطاء التي تقوم بها بشكل شائع بالإضافة إلى سجل لمقدار الوقت الذي تقضيه في كتابة الشفرة، واختبار الشفرة، وتصحيح الأخطاء.

مصادر إضافية[1]

تجبرني قوانين الحقائق الإخبارية في الحقيقة على الكشف عن أن العديد من الكتب الأخرى تغطي الاختبار بمزيد من العمق أكثر مما يفعل هذا الفصل. تُناقش الكتب المخصصة للاختبار نظام واختبار الصندوق الأسود، التي لم تتم مناقشتها في هذا الفصل. كما أنهم يتعمقون أكثر في موضوعات المطورين. وتُناقش هذه الكتب الطرق الرسمية مثل الرسوم البيانية للتأثير، وخصوصيات وعموميات تأسيس منظمة اختبار مستقلة.

الاختبار

Kaner, Cem, Jack Falk, and Hung Q. Nguyen. Testing Computer Software, 2d ed. New York, NY: John Wiley & Sons, 1999.

كانير، جيم، جاك فالك، وهونغ كيو نغوين. اختبار برامج الحاسوب، الإصدار الثاني، نيويورك: جون وايلي وأولاده، 1999. ربما هذا هو أفضل كتاب حالي في اختبار البرمجيات. وهو الأكثر قابلية للتطبيق على تطبيقات الاختبار التي سيتم توزيعها على قاعدة عملاء واسعة الانتشار، مثل مواقع الويب ذات الحجم الكبير وتطبيقات shrink-wrap، وأيضًا مفيدة بشكل عام.

Kaner, Cem, James Bach, and Bret Pettichord. Lessons Learned in Software Testing. New York, NY: John Wiley & Sons, 2002.

كانير، جيم، جاك فالك، وبريت بيتيتشورد. الدروس المستفادة في اختبار البرمجيات. نيويورك: جون وايلي وأولاده، 2002. هذا الكتاب هو تكملة جيدة لكتاب " اختبار برامج الحاسب"، الإصدار الثاني. تم تنظيم هذا الكتاب في 11 فصلًا تضم 250 درسًا مُتعلمة عن طريق المؤلفين.

Tamre, Louise. Introducing Software Testing. Boston, MA: Addison-Wesley, 2002.

تامري، لويز. مقدمة اختبار البرمجيات. بوسطن، ماساتشوستس: أديسون ويسلي، 2002. هذا كتاب اختبار مُستهدف للمطورين الذين يحتاجون إلى فهم الاختبار. على نقيض عنوان الكتاب، يتعمق الكتاب بعض الشيء في تفاصيل الاختبار المفيدة حتى للمختبرين ذوي الخبرة.

Whittaker, James A. How to Break Software: A Practical Guide to Testing. Boston, MA: Addison-Wesley, 2002.

ويتاكر، وجيمس. كيفية كسر البرمجيات: دليل عملي لاختبار. بوسطن، ماساتشوستس: أديسون ويسلي، 2002. يسرد هذا الكتاب 23 هجومًا يمكن أن يستخدمها المختبرون لجعل البرمجية تفشل، وتقدم أمثلة لكل هجوم

[1] cc2e.com/2203

باستخدام حزم البرامج الشائعة. يمكنك استخدام هذا الكتاب كمصدر أساسي للمعلومات حول الاختبار أو، نظرًا لأن منهجها مميز، يمكنك استخدامه لاستكمال كتب الاختبار الأخرى.

Whittaker, James A. "What Is Software Testing? And Why Is It So Hard?" IEEE Software, January 2000, pp. 70–79.

ويتاكر، جيمس أ. "ما هو اختبار البرمجيات؟ ولماذا هو صعب للغاية؟ "برمجياتIEEE، كانون الثاني 2000، الصفحات 70–79. تعد هذه المقالة مقدمة جيدة لقضايا اختبار البرمجيات وتشرح بعض التحديات المرتبطة ببرمجيات الاختبار الفعالة.

Myers, Glenford J. The Art of Software Testing. New York, NY: John Wiley, 1979.

هذا هو الكتاب الكلاسيكي لاختبار البرمجيات وما زال مطبوعًا (رغم أنه مكلف للغاية). محتويات الكتاب واضحة: اختبار التقييم الذاتي؛ علم النفس والاقتصاد في اختبار البرامج؛ عمليات التفتيش على البرامج، والمواضيع الإرشادية والمراجعات؛ تصميم حالة الاختبار؛ اختبار الصف؛ اختبار من الدرجة العليا؛ عملية التصحيح؛ أدوات الاختبار وتقنيات أخرى. هذا الكتاب قصير (177 صفحة) وقابل للقراءة. يساعدك الاختبار في البداية على البدء في التفكير كمختبر ويوضح عدد الطرق التي يمكن بها كسر جزء من الشفرة.

سقالات الاختبار

Bentley, Jon. "A Small Matter of Programming" in Programming Pearls, 2d ed. Boston, MA: Addison-Wesley, 2000.

بنتلي، جون. "مسألة صغيرة من البرمجة" في درر البرمجة، الإصدار الثاني. بوسطن، ماساتشوستس: أديسون ويسلي، 2000. يتضمن هذا المقال العديد من الأمثلة الجيدة على سقالات الاختبار.

Mackinnon, Tim, Steve Freeman, and Philip Craig. "Endo-Testing: Unit Testing with Mock Objects," eXtreme Programming and Flexible Processes Software Engineering - XP2000" Conference, 2000.

هذه هي الورقة الأصلية لمناقشة استخدام كائنات مزيفة لدعم اختبار المطور.

Thomas, Dave and Andy Hunt. "Mock Objects," IEEE Software, May/June 2002.

توماس، ديف واندي هانت. "الكائنات المُزيفة"، برمجيات IEEE، أيار / حزيران 2002. هذه مقدمة قابلة للقراءة لاستخدام كائنات مزيفة لدعم اختبار المطور.

www.junit.org.[1] يوفر هذا الموقع الدعم للمطورين لاستخدام Junit. يتم توفير مصادر مماثلة في cppunit.sourceforge.net، وnunit.sourceforge.net.

اختبار التطوير الأول

Beck, Kent. Test-Driven Development: By Example. Boston, MA: Addison-Wesley, 2003.

بيك، كنت. التطوير المُقاد بالاختبار: حسب المثال. بوسطن، ماساتشوستس: أديسون ويسلي، 2003. يصف بيك خصوصيات وعموميات "التطوير المُقاد بالاختبار"، وهو منهج تطوير يتميز بكتابة حالات الاختبار أولاً ثم كتابة الشفرة لتلبية حالات الاختبار. على الرغم من نبرة بيك الإنجيلية في بعض الأحيان، فإن النصيحة سليمة، والكتاب قصير إلى حد كبير. يحتوي الكتاب على مثال تشغيل واسع بشفرة حقيقية.

المعايير ذات الصلة

معيار لاختبار وحدة البرمجيات:

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing.

معيار لتوثيق اختبار البرمجيات:

IEEE Std 829-1998, Standard for Software Test Documentation.

معيار لخطط ضمان جودة البرمجيات:

IEEE Std 730-2002, Standard for Software Quality Assurance Plans.

- **لائحة اختبار2:**
- هل لدى كل متطلب يتم تطبيقه على صف أو إجرائية حالة اختبار خاصة به؟
- هل لدى كل عنصر من التصميم الذي ينطبق على الصف أو الإجرائية، حالة اختبار خاصة به؟
- هل تم اختبار كل سطر من الشفرة بحالة اختبار واحدة على الأقل؟ هل تم التحقق من هذا بحساب العدد الأصغري من الاختبارات الضرورية لتنفيذ كل سطر من الشفرة؟
- هل تم اختبار كل مسارات تدفق المعطيات مُعرف- مُستخدم بحالة اختبار واحدة على الأقل؟
- هل تم فحص الشفرة لمخططات تدفق معطيات، من غير المحتمل أن تكون صحيحة، مثل مُعرف – مُعرف، ومُعرف – مُخرج، ومُعرف – مقتول؟
- هل تم استخدم قائمة بالأخطاء الشائعة لكتابة حالات الاختبار للكشف عن الأخطاء التي حدثت بشكل متكرر في الماضي؟
- هل تم اختبار كل الحدود البسيطة: الحد الأعلى، والحد الأدنى، وحدود الخطوة الواحدة؟

[1] cc2e.com/2217
[2] cc2e.com/2210

- هل تم اختبار الحدود المُركبة- أي دمج من معطيات الدخل التي قد ينتج عنها متغير محسـوب صغير جدًا أو كبير جدًا؟
- هل تتحقق حالات الاختبار من النوع المعطيات ذات النوع الخاطئ- على سـبيل المثال، عدد سلبي من العمال في برنامج كشف رواتب؟
- هل تم اختبار جميع القيم مقبولة الممثَّلة؟
- هل تم اختبار الحد الأدنى للإعدادات الطبيعية (العادية)؟
- هل تم اختبار الحد الأعلى للإعدادات الطبيعية (العادية)؟
- هل يوجد توافق مع البيانات القديمة التي تم اختبارها؟
- والأجهزة القديمة، والإ صدارات القديمة من أنظمة التـشغيل، والواجهات مع إ صدارات قديمة من برمجيات مُختبرة؟
- هل يجعل الاختبار التحقق باليد سهل؟

نقاط مفتاحية

- إن اختبار بواسطة المطور هو جزء أساسي من استراتيجية اختبار كاملة. والاختبار المستقل أيضًا مهم، ولكنه خارج مجال هذا الكتاب.
- تأخذ كتابة حالات الاختبار قبل كتابة الشـفرة نفس كمية الوقت والجهد الذي يطلبه كتابة حالات الاختبار بعد كتابة الشفرة، ولكنه يُقصّر دورات كشف- معالجة- تصحيح الأخطاء.
- حتى إذا أخذنا في الاعتبار الأنواع العديدة من الاختبارات المتاحة، فإن الاختبار هو جزء واحد فقط من برنامج جيد لجودة البرمجيات. إن مناهج التطوير عالية الجودة، بما فيها إنقاص العيوب في المتطلبات والتصميم، على الأقل بنفس القدر من الأهمية. وممارسات التطوير التعاونية هي أيضًا على الأقل فعالة في الكشـف عن الأخطاء مثل الاختبار، وتكشـف هذه الممارسـات عن أنواع مختلفة من الأخطاء.
- يمكنك توليد العديد من حالات الاختبار بشـكل حتمي باسـتخدام اختبار الأسـاس، وتحليل تدفق المعطيات، وتحليل الحدود، وصـفوف المعطيات السـيئة، وصـفوف المعطيات الجيدة. ويمكنك توليد حالات اختبار إضـافية باسـتخدام تخمين الخطأ. تميل الأخطاء إلى أن تتجمع في بضـعة صـفوف وإجرائيات مُعرضة للخطأ. جد تلك الشفرة المعرضة للخطأ، وأعد تصميمها، ومن ثم أعد كتابتها.
- تميل بيانات الاختبار إلى الحصـول على كثافة خطأ أعلى من الشـفرة التي يتم اختبارها. لأن ملاحقة أخطاء كهذه تهدر الوقت بدون تحسين الشفرة، أخطاء بيانات الاختبار أكثر إزعاجًا من أخطاء البرمجة. تجنبهم بتطوير اختباراتك بنفس العناية والحذر عند تطوير شفرتك.

- الاختبار الآلي مُفيد بشكل عام وهو أساسي من أجل اختبار التراجع.
- من أجل التشـــغيل الطويل، أحســـن طريقة لتحســـين عملية الاختبار هي بجعلها مُنتظمة، وقياســـها، واستخدام ما تعلمته لتحسينها.

التصحيح

المحتويات[1]

مواضيع ذات صلة

- المنظر الطبيعي لجودة البرمجيات: الفصل 20
- اختبار المطور: الفصل 22
- إعادة التصنيع: الفصل 24

التصحيح هو عملية تحديد السبب الجذر لخطأ وتصحيحه. إنه يتباين عن الاختبار،[2] والذي هو عملية اكتشاف الأخطاء في البداية. في بعض المشاريع يشغل *التصحيح* 50 بالمئة من وقت التطوير الكلي. بالنسبة للعديد من المبرمجين، *التصحيح* هو الجزء الأصعب من البرمجة.

لا يجب على *التصحيح* أن يكون الجزء الأصعب. إذا اتبعت النصيحة في هذا الكتاب، سيكون لديك أخطاء أقل للتصحيح. معظم العيوب التي ستكون لديك هي سهوات ثانوية وأخطاء كتابية، سهلة الاكتشاف بالنظر إلى لائحة الشفرة المصدرية أو بالخَطو عبر الشفرة *بالمصحح*. من أجل الثغرات الأصعب الباقية، يشرح هذا الكتاب كيف تجعل *التصحيح* أسهل مما هو عليه عادةً.

---

[1] cc2e.com/2361

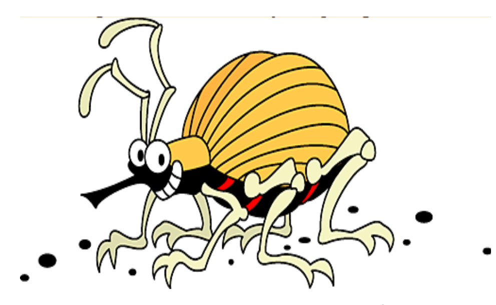
هامش تم اعتماد الاصطلاح:*debugging* تصحيح (بالمائل) لتفريقها عن correcting. *debugger المصحح*...

[2] التصحيح أصعب بمرتين من كتابة الشفرة في المقام الأول. لذا، إذا كنت تكتب الشفرة بقدر الذكاء الممكن، فإنك، حسب التعريف، لست ذكياً كفاية لتصحححه. —براين دبليو. كيرنيغان

23.1 نظرة عامة على قضايا *التصحيح*

كان الأميرال الخلفي المتأخر غريس هوبر، المشارك باختراع كوبول، يقول دائماً أن الكلمة "حشرة" (bug) يرجع تاريخها في البرمجيات إلى أول حاسوب رقمي واسع النطاق، مارك 1 (آي تربل إي 1992). تعقّب المبرمجون القصر في الدارة ليكتشفوا حضور قملة كبيرة وجدت طريقها إلى الحاسوب، ومنذ ذلك الوقت، أُلقي اللوم في مشاكل الحاسوب على "الحشرات." خارج البرمجيات، يرجع تاريخ الكلمة "حشرة" على الأقل إلى توماس أديسون، الذي اسُتشهد أنه استخدمها بقِدم 1878 (تنر 1997).

الكلمة "حشرة" كلمة جذابة وتستحضر صورا مثل هذه:



حقيقة عيوب البرمجيات، على أي حال، هو أن "الحشرات" ليست كائنات حية تتسلل إلى شفرتك عندما تنسى أن ترشّ مضاد حشرات. إنها أخطاء. الحشرة (الثغرة) في البرمجيات تعني أن المبرمج ارتكب خطأ. نتيجة الخطأ ليست كالصورة الكيّسة في الأعلى. إنها أكثر شبهاً بملاحظة مثل هذه:



في سياق هذا الكتاب، تتطلب الجودة التقنية أن تُسمَّى الأخطاء في الشفرة "أخطاء" (error)أو "عيوب" (defects) أو "خِلال"(faults).

قاعدة *التصحيح* في جودة البرمجية

مثل الاختبار، ليس *التصحيح* طريقة لتحسين جودة برمجيتك بحد ذاته؛ إنه طريقة لتشخّص العيوب. يجب حتماً أن تبنى جودة البرمجية من البداية. أفضل طريقة لتبني منتج ذو جودة هي أن تتطور المتطلبات بحذر وتصمم بشكل جيد وتستخدم تطبيقات كتابة الشفرة عالية الجودة. *التصحيح* هو الملاذ الأخير.

الاختلافات في أداء *التصحيح*



لمَ الحديث عن *التصحيح*؟ ألا نعلم كلنا كيف *نصحح*؟

لا، ليس كلنا يعلم كيف *يصحح*. وجدت دراسات عن المبرمجين ذوي الخبرة، تفاوتاً بنسبة حوالي 20 إلى 1 في الوقت الذي يستغرقه المبرمجون الخبراء ليجدوا نفس العيوب التي وجدها مبرمجون غير خبراء. أكثر بعد، يجد بعض المبرمجين عيوب أكثر ويقوم بتصحيحات أدق. إليك نتائج دراسة تقليدية فحصت كيف صحح مبرمجون محترفون بخبرة أربع سنين على الأقل، برنامجاً ب 12 عيب بفعالية:

	أسرع ثلاثة مبرمجين	أبطأ ثلاثة مبرمجين
وقت التصحيح الوسطي (بالدقائق)	5.0	14.1
العدد الوسطي للعيوب التي لم تكتشف	0.7	1.7
العدد الوسطي للعيوب التي صنعت عند تصحيح العيوب	3.0	7.7

المصدر: بعض الأدلة النفسية على كيفية *تصحيح* الناس لبرامج الحاسوب " Some Psychological Evidence on How People Debug Computer Programs" (غولد 1975)

كان المبرمجون الثلاثة الأفضل في *التصحيح* قادرين على إيجاد العيوب بحوالي ثلث الوقت وحشروا حوالي خُمسين فقط من الأخطاء الجديدة بالمقارنة مع الثلاثة الأسوأ. وجد المبرمجون الأفضل كل العيوب ولم يحشروا أي عيوب جديدة في تصحيحها. غفل الأسوأ عن 4 من 12 عيب وحشروا 11 عيب جديد في تصحيح ال 8 عيوب التي وجدوها.

لكن هذه الدراسة لا تخبر القصة الكاملة حقيقةً. بعد الدور الأول من *التصحيح*، لازال لدى المبرمجين الثلاثة الأسرع 3.7 عيب باق في شفرتهم ولازال لدى الأبطأ 9.4 عيب. ولا واحدة من المجموعتين قامت *بالتصحيح* حتى الآن. أتساءل ماذا سيحدث إذا طبّقت نفس معدليّ "الاكتشاف والإصلاح السيء" إلى دورات *التصحيح* الإضافية. نتائجي ليست صالحة إحصائياً، لكنها لا تزال ممتعة. عندما طبّقت نفس المعدلين لدورات التصحيح الإضافية مراراً حتى تبقّى لدى كل مجموعة أقل من نصف العيوب، تطلبت أسرع مجموعة مجموع ثلاث دورات تصحيح، بينما تطلبت المجموعة الأبطأ 14 دورة تصحيح. محاكماً في عقلي أن كل دورة للمجموعة الأبطأ ستأخذ حوالي 13 مرة طولاً لتصحح برامجها بشكل تام بالمقارنة مع ما ستأخذه المجموعة الأسرع، حسب استقرائي غير العلمي لهذه الدراسة. هذا التباين الواسع أكد بعدة دراسات أخرى (غيلب 1977، كورتيس 1981).

بالإضافة إلى تقديمها رؤية في *التصحيح*، تدعم هذه الشهادة **المبدأ العام في جودة البرمجيات**[1]: تحسين الجودة يخفّض كُلف التطوير. وجد المبرمجون الأفضل معظم العيوب، ووجدوا العيوب بسرعة أكبر، وقاموا

[1] **إشارة مرجعية** لتفاصيل حول العلاقة بين الجودة والكلفة، انظر القسم 20. 5، "المبدأ العام في جودة البرمجيات".

بالتعديلات الصحيحة غالباً. ليس عليك أن تختار بين الجودة والكلفة والوقت-كل منها تمسك بيد الأخرى ويذهبن معاً.

العيوب باعتبارها فُرَصاً

ماذا يعني أن يكون لديك عيب؟ بادعائك أنك لا تريد أن يمتلك برنامجك عيباً، هذا يعني أنك لم تفهم بشكل كامل ما يقوم به البرنامج. الفكرة: أنك غير فاهم لما يقوم به البرنامج، هي مقلقة. بعد كل ذلك، إذا أنشأت البرنامج، ينبغي أن يقوم بما تدّعي. إن لم تعرف بالضبط ماذا تطلب من الحاسوب أن يفعل، فإنك على بعد خطوة صغيرة فقط من مجرد محاولة أشياء مختلفة حتى يبدو أن شيء ما قد اشتغل -هذا يعني، البرمجة عن طريق التجريب والخطأ. وإذا كنت تبرمج عن طريق التجريب والخطأ، فإن العيوب مضمونة الحدوث. إنك لا تحتاج أن تتعلم كيف تصلح العيوب؛ إنك تحتاج أن تتعلم كيف تتجنبها في المقام الأول.

معظم الناس لديهم قابلية ارتكاب الأخطاء، على كلّ، وقد تكون مبرمجاً ممتازاً وببساطة قمت بسهوة متواضعة. إن كانت هذه هي الحالة، فإن خطأ في برنامجك يؤمن لك فرصة قوية لتتعلم العديد من الأشياء. تستطيع أن:

تتعلم عن البرنامج الذي تعمل عليه لديك شيء ما لتتعلمه عن البرنامج لأنه إن كنت تعرفه مسبقاً بشكل تام، فما كان ليمتلك عيباً. قد قمت بتصحيحه مسبقاً.

تتعلم ما هي أنواع الأخطاء التي ترتكبها[1] إن كتبت البرنامج، حشرتَ العيب. لن يكون في كل يوم بقعة مضيئة تكشف الضعف بوضوح ساطع، لكن مثل هذا اليوم فرصة، لذا اغتنمه. حالما تجد خطأ، اسأل نفسك كيف ولم ارتكبته. كيف يمكن أن تجده بسرعة أكبر؟ كيف يمكن أن تحول دونه؟ هل تحتوي الشفرة أخطاء مثله تماماً؟ هل تستطيع تصحيحها قبل أن تسبب المشاكل من تلقاء نفسها؟

تتعلم عن جودة شفرتك من وجهة نظر شخص عليه أن يقرأها عليك أن تقرأ شفرتك لتجد العيوب. هذه فرصة لتنظر بحذر إلى جودة شفرتك. هل هي سهلة القراءة؟ كيف يمكن أن تكون أفضل؟ استخدم اكتشافاتك لتعيد تصنيع شفرتك الحالية أو لتحسن الشفرة التي ستكتبها لاحقاً.

تتعلم عن كيفية إصلاح العيوب بالإضافة إلى تعلمك كيف تكتشف العيوب، تستطيع أن تتعلم كيف تصلحها. هل قمت بأسهل تصحيح ممكن بواسطة ضمادات *goto* ومكياج "الحالة الخاصة" والتي تغير العرض وليس المشكلة؟ أم هل قمت بتصحيحات نظامية، مشترطاً تشخيصاً دقيقاً وواصفاً العلاج للب المشكلة؟

---

[1] **اقرأ أيضاً** لتفاصيل عن تطبيقات ستساعدك في تعلم ما هي أنواع الأخطاء التي تميل شخصياً إليها، انظر *A Discipline for Software Engineering* (هامفري 1995).

باعتبار كل الأشياء، *التصحيح* هو تربة غنية بشكل استثنائي لتزرع بذور تحسيناتك الخاصة. إنه المكان حيث كل طرق البناء تلتقي: إمكانية القراءة، والتصميم، وجودة الشفرة-اذكرها أنت. هذا هو المكان حيث يقوم بناء شفرة جيدة بدفع كل ما يترتب عليه، خصوصاً إذا قمت به بشكل جيد كفاية حتى لا يتوجب عليك *التصحيح* مرات كثيرة.

نهجٌ غير مجدٍ

لسوء الحظ، صفوف البرمجة في الكليات والجامعات نادراً ما قدم تعليمات في *التصحيح*. على الرغم من كون تعليمي في علوم الحاسوب ممتازاً، فإن نهاية نصيحة *التصحيح* التي تلقيتها كانت "ضع عبارات طباعة في البرنامج لتجد العيوب." هذا غير ملائم. إذا كانت الخبرات التعليمية للمبرمجين الآخرين مثل خبرتي، فإن عدد هائل من المبرمجين سيجبرون على إعادة اختراع مفاهيم *تصحيحية* لوحدهم. يا له من ضياع!

المرشد الشيطاني في التصحيح

في تصور دانتي لجهنم، الحلقة الدنيا محجوزة لإبليس نفسه. في الآونة الأخيرة، [1]تم الاتفاق على أن الشيطان (يسمونه أيضا "العبث القديم" `Old Scratch`) يتشارك الدائرة الدنيا مع المبرمجين الذين لم يتعلموا كيف *يصححون* بفعالية. إنه يعذب المبرمجين بجعلهم يستخدمون نُهج *التصحيح* هذه:

جِد العيوب عن طريق التخمين لتجد عيب، بعثر عبارات طباعة بشكل عشوائي خلال البرنامج. تفحص الخرج لترى أين يكون العيب. إن لم تجد العيب عن طريق عبارات الطباعة، حاول تغيير أشياء في البرنامج حتى يبدو أن شيء ما اشتغل. لا تعمل نسخ احتياطية للإصدارات الأصلية للبرنامج، ولا تحافظ على سجل التغييرات التي قمت بها. البرمجة أكثر إثارة عندما لا تعرف تماماً ما يقوم به البرنامج. جهّز مونة الكولا والحلويات لأنك ستكون في ليلة طويلة أمام الشاشة.

لا تضيع الوقت بمحاولة فهم المشكلة من المرجح أن المشكلة سخيفة، ولا تحتاج ان تفهمها بشكل كامل لتصلحها. ببساطة، أن تجدها كافٍ.

أصلح الخطأ بالإصلاح البائن أكثر عادةً يكون من الجيد أن تصلح مشكلة محددة رأيتها، بدلاً من تضييع الكثير من الوقت بصنع تصحيح كبير وطموح سيؤثر بكل البرنامج. هذا مثال مثالي:

[1] لا يستخدم المبرمجون دائماً البيانات المتاحة ليوثقوا استنتاجاتهم. إنهم يقومون بإصلاحات أصغرية وغير منطقية، وهم غالباً لا يتراجعون عن الإصلاحات غير الصحيحة. --إرس فسي

التصحيح

```
x = Compute( y )
if ( y = 17 )
x = $25.15
```

Compute() لاتعمل من اجل y=17، لذا أصلحها--

من يحتاج أن يحفر كل الطريق إلى ()Compute من أجل مشكلة غامضة مع القيمة 17 عندما تستطيع أن تكتب الحالة الخاصة فقط لها في المكان البائن؟

التصحيح عن طريق الخرافات

أجّر إبليس قسماً من جهنم للمبرمجين الذين *يصححون* عن طريق الخرافات. كل مجموعة لديها مبرمج لديه مشاكل غير منتهية مع آلات عفريتية، وعيوب المترجم السحرية، وعيوب اللغة المخفية التي تظهر عندما يكون القمر مكتملاً، والبيانات السيئة، وخسارة التغييرات الهامة، ومنسق ممسوس يحفظ (save) البرامج بطريقة غير صحيحة- أكمل أنت. هذه هي "البرمجة عن طريق الخرافة."

إذا كان لديك مشكلة مع برنامج أنت قمت بكتابته، فإنه غلطك. إنه ليس غلط الحاسوب، وإنه ليس غلط المترجم. لا يقوم البرنامج بشيء ما مختلف، في كل مرة يُشغَّل بها. إنه لم يكتب نفسه، أنت كتبته، لذا تحمّل مسؤولية ذلك.



حتى ولو ظهر خطأ في البداية على أنه ليس غلطك، فإنه لصالحك بقوة أن تفترض أنه غلطك. يساعدك هذا الافتراض على *التصحيح*. من الصعب جداً أن تجد عيباً في شفرتك عندما تبحث عنه؛ إنه حتى أصعب عندما تفترض أن شفرتك خالية من الأخطاء. يحسّن افتراض أن الخطأ غلطك من مصداقيتك أيضاً. إذا ادعيّت أن الخطأ ظهر من شفرة شخص آخر، سيعتقد المبرمجون الآخرون أنك قد فحصت المشكلة بحذر. إذا افترضت أن الخطأ هو خطأك، فإنك تتجنب الإحراج الناتج عن سحب قولك أمام العموم لاحقاً عندما تكتشف أنه كان عيبك بعد كل شيء.

23.2 إيجاد العيوب

يتألف *التصحيح* من إيجاد العيوب وإصلاحها. إيجاد العيب-وفهمه-هو بالعادة 90 بالمئة من العمل.

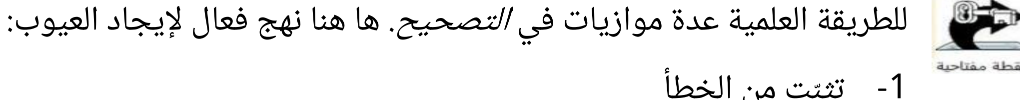
لحسن الحظ، ليس عليك أن تقيم معاهدة مع إبليس لتجد نهجاً في *التصحيح* أفضل من التخمين العشوائي. *التصحيح* عن طريق التفكير بالمشكلة أكثر فعالية وإمتاع بكثير من *التصحيح* الشيطاني.

افترض أنك سُئلت بأن تبيّن غوامض جريمة. أيهما سيكون ممتعاً أكثر: أن تذهب من باب إلى باب في المقاطعة، فاحصاً ادعاء كل متهم عن ليلة 17 تشرين الأول، أو اكتشاف القليل من أطراف الخيوط ثم الاستدلال إلى هوية المجرم؟ معظم الناس يفضلون الاستدلال على هوية الشخص، ومعظم المبرمجين يجدون النهج الفكري في *التصحيح* أكثر إرضاء. وحتى أفضل، لا يقوم المبرمجون الفعالون، الذي *يصححون* في جزء من عشرين جزء من الوقت الذي يستخدمه المبرمجون غير الفعالون، بتخمين عشوائي عن كيفية إصلاح البرنامج. إنه يستخدمون الطريقة العلمية-وهي: عملية الاكتشاف والتوضيح الضرورية للاستقصاء العلمي.

الطريقة العلمية في *التصحيح*

توجد هنا الخطوات التي تسير عبرها عندما تستخدم الطريقة العلمية التقليدية:

1. اجمع البيانات عبر تجارب متكررة.
2. شكّل فرضيّة تحسب حساب البيانات ذات الصلة
3. صمم تجربة لتثبت أو تنفي الفرضية.
4. أثبت أو انفي الفرضية
5. كرر متى احتجت



للطريقة العلمية عدة موازيات في *التصحيح*. ها هنا نهج فعال لإيجاد العيوب:

1- تثبّت من الخطأ

2- حدد موقع مصدر الخطأ ("الغلط")

أ. اجمع البيانات التي أنتجت الغلط.

ب. حلل البيانات التي جُمعت، وشكّل فرضية عن العيب

ج. حدد كيفية إثبات أو نفي الفرضية، إما باختبار البرنامج أو بامتحان الشفرة.

د. أثبت أو انفي الفرضية باستخدام الإجراء المعرف في الخطوة 2.

3- أصلح العيب

4- اختبر الإصلاح

5- ابحث عن أخطاء مشابهة.

الخطوة الأولى مشابهة لمقابلتها في الطريقة العلمية في أنها تعتمد على قابلية التكرار. من الأسهل تشخيص العيب إذا استطعت التثبت منه-هذا يعني، تجعله يحدث بشكل موثوق. تستخدم الخطوة الثانية خطوات

الطريقة العلمية. فإنك تجمع بيانات الاختبار التي تكشف العيب، وتحلل البيانات التي أنتجت، وشكّل فرضية عن مصدر الخطأ. ثم تصمم حالة اختبار أو تفحص لتقيّم الفرضية، وإما تعلن نجاحاً (تبعاً لإثبات فرضيتك) أو تجدد مساعيك، كما هو مناسب. بعدما تبرهن فرضيتك، تصلح العيب، وتختبر الإصلاح، وتبحث في شفرتك عن أخطاء مشابهة.

دعنا نلقي نظرة على كل واحدة من الخطوات بربطها بمثال. افترض أنه لديك برنامج قاعدة بيانات موظفين. وفيه خطأ متقطع. يُفترض أن يطبع البرنامج لائحة بالموظفين وضرائب الدخل الخاصة بهم بترتيب أبجدي. إليك جزء من الخرج:

```
$5,877      Formatting, Fred Freeform
$1,666      Global, Gary
$10,788     Modula, Mildred
$8,889      Many-Loop, Mavis
$4,000      Statement, Sue Switch
$7,860      Whileloop, Wendy
```

الخطأ هو أن `Many-Loop, Mavis` و`Modula, Mildred` ليستا مرتبتين.

التثبت من الخطأ

إن لم يحدث العيب بشكل موثوق، فإنه من المستحيل أن يُشخَّص. أن تجعل عيباً متقطعاً يحدث بشكل قابل للتوقع واحد من المهام الأكثر تحدياً في *التصحيح*.

عادةً ما يكون الخطأ الذي لا يحدث بشكل مُتوقَّع خطأ تهيئة[1]، أو مشكلة في التوقيت، أو مشكلة مؤشر متدلٍّ. إذا كانت حسابات مجموع صحيحة أحياناً وخاطئة أحياناً، فإن المتحول المُضمَّن في الحسابات قد لا يكون مهيأً بشكل صحيح-معظم الأوقات تصدف أن يبدأ ب 0. إن كانت المشكلة ظاهرة غريبة وغير قابلة للتوقع وكنت *تستخدم* المؤشرات، تقريباً من المؤكد أن لديك مؤشر غير مهيأ أو أنك *تستخدم* مؤشر بعد إلغاء تخصيص الذاكرة التي يؤشر إليها.

عادةً ما يتطلب التثبت من خطأ أكثر من إيجاد حالة اختبار تؤدي إلى الخطأ. إنه يتضمن تضييق حالة الاختبار إلى أبسط شكل لا يزال يؤدي إلى الخطأ. الغرض من تبسيط حالة الاختبار هو أن تجعلها بسيطة جداً بحيث يغير تغيُّر أي من جوانبها من سلوك الخطأ. ومن ثم، بتغيير حالة الاختبار بحذر ومراقبة سلوك البرنامج تحت

[1] **إشارة مرجعية** لتفاصيل حول استخدام المؤشرات بأمان، انظر القسم 13. 2، "المؤشرات"

الظروف المدارة، تستطيع تشخيص المشكلة. إذا كنت تعمل في منظمة لديها فريق فحص مستقل، أحياناً يكون من عمل الفريق أن يصنع حالة الاختبار بسيطة. في معظم الأحيان، إنه عملك.

لتبسيط حالة الاختبار، احضر الطريقة العلمية إلى اللعبة مجدداً. افترض أنه لديك 10 عوامل، مستخدمة في تجميعة، تؤدي إلى الخطأ. شكّل الفرضية عن أي العوامل غير متصلة بإنتاج الخطأ. غيّر العوامل المفترَضة أنها غير ذات صلة، وأعد تشغيل حالة الاختبار. بعدها تستطيع أن تحاول تبسيط الاختبار أكثر. إذا لم تحصل على الخطأ، فإنك تكون قد نفيت الفرضية المحددة وأصبحت تعرف أكثر مما كنت تعرف من قبل. قد تكون الحالة أنه لا يزال تغيير مختلف بشكل يصعب تمييزه ينتج الخطأ، لكنك تعرف تغيير محدد على الأقل لا يقوم بذلك.

في مثال ضرائب الموظفين، عندما اشتغل البرنامج في البداية، *Many-Loop, Mavis* ذكرت بعد *Modula, Mildred*. عندما اشتغل البرنامج المرة الثانية، على أي حال، كانت اللائحة جيدة.

Formatting, Fred Freeform	$5,877
Global, Gary	$1,666
Many-Loop, Mavis	$8,889
Modula, Mildred	$10,788
Statement, Sue Switch	$4,000
Whileloop, Wendy	$7,860

إنها ليست كذلك بعد أن أُدخلت *Fruit-Loop, Frita* وظهرت في مكان غير صحيح بحيث تتذكر أن *Modula, Mildred* قد أدخلت مباشرة قبل ظهورها في موضع خاطئ أيضاً. الذي هو شاذ في كلا الحالتين هو أن الإدخال كان بشكل إفرادي. عادةً، يُدخل الموظفون في مجموعات.

قد تفترض: المشكلة مرتبطة بالقيام بإدخال موظف مفرد جديد. إن كان هذا صحيحاً، تشغيل البرنامج مرة ثانية ينبغي أن يضع *Fruit-Loop, Frita* في المكان الصحيح. هذه هي النتيجة في التشغيل الثاني:

Formatting, Fred Freeform	$5,877
Fruit-Loop, Frita	$5,771
Global, Gary	$1,666
Many-Loop, Mavis	$8,889
Modula, Mildred	$10,788
Statement, Sue Switch	$4,000
Whileloop, Wendy	$7,860

هذا التشغيل الناجح يدعم الفرضية. لتؤكدها، عليك أن تحاول إضافة عدة موظفين جدد، واحد في كل مرة، لترى إن كانوا سيظهرون في ترتيب خاطئ وإن كان الترتيب سيتغير في التشغيل اللاحق.

عيّن موقع مصدر الخطأ

تعيين موقع مصدر الخطأ أيضاً يستدعي استخدام الطريقة العلمية. قد تتوقع أن العيب نتيجة مشكلة محددة، لنقل خطأ "خارجاً من مرة واحدة". تستطيع بعدها أن تغيّر الوسطاء التي تتوقع أنها مسببة للمشكلة-واحد تحت الحد، وواحد على الحد، وواحد فوق الحد- وتحدد إن كانت فرضيتك صحيحة.

في المثال الجاري، قد يكون مصدر المشكلة عيب "خارجاً من واحدة" والذي يحدث عندما تضيف موظف جديد واحد لكن ليس عندما تضيف اثنين أو أكثر. بفحصك للشفرة، لا تجد عيب "خارجاً من واحدة" واضح. بلجوئك إلى المخطط ب، تقوم بتشغيل حالة اختبار بموظف جديد واحد لترى إن كانت هذه هي المشكلة. تضيف *Hardcase, Henry* كموظف وحيد وتفترض أن سجله سيكون خارج الترتيب. إليك ما تجد:

Formatting, Fred Freeform	$5,877
Fruit-Loop, Frita	$5,771
Global, Gary	$1,666
Hardcase, Henry	$493
Many-Loop, Mavis	$8,889
Modula, Mildred	$10,788
Statement, Sue Switch	$4,000
Whileloop, Wendy	$7,860

سطر *Hardcase, Henry* هو تماماً حيث ينبغي أن يكون، والذي يعني أن فرضيتك الأولى خاطئة. لم تكن المشكلة ببساطة ناتجة عن إضافة موظف واحد في كل مرة. إنها إما مشكلة أكثر تعقيد أو شيء ما مختلف تماماً.

بفحص خرج تشغيل الفحص مجدداً، تلاحظ أن *Fruit-Loop, Frita* و*Many-Loop, Mavis* هما الاسمين الوحيدين الذين يحتويان شحطات. *Fruit-Loop* كانت خارج الترتيب عندما أُدخلت في البداية، لكن *Many-Loop* لم تكن، أكانت؟ على الرغم من أنك لا تملك خرج مطبوع من المدخل الأصلي، في الخطأ الأصلي بدت *Modula, Mildred* أنها خارج الترتيب، لكنها كانت مباشرة بعد *Many-Loop*.ربما كانت *Many-Loop* خارج الترتيب و*Modula* كانت تمام.

تفترض مجدداً: تظهر المشكلة من أسماء تحوي شحطات، وليس من الأسماء التي تُدخل على إفراد.

لكن كيف لذلك أن يحسب حساب حقيقة أن المشكلة تظهر فقط بعد أول مرة يُدخل فيها الموظف؟ تنظر إلى الشفرة وتجد إجرائيتين مختلفتين مستخدَمتين. واحدة مستخدمة عندما يُدخل موظف، والأخرى عندما تُخزّن البيانات. نظرة أقرب على الإجرائية المستخدمة عندما يُدخل موظف لأول مرة تُظهر أنه ليس من المفترض أن ترتّب البيانات بشكل كامل. إنها فقط تضع البيانات في ترتيب تقريبي لتسرّع ترتيب إجرائية الحفظ. وهكذا، المشكلة هي أن البيانات تُطبع قبل أن تُرتب. تظهر المشكلة من الأسماء الحاوية على شحطات لأن إجرائية الترتيب التقريبي لا تتعامل مع التفاصيل الدقيقة مثل محارف الترقيم. الآن، تستطيع أن تصقّي الفرضية أكثر، بعد.

تفترض لمرة أخيرة: لا تُرتّب الأسماء الحاوية على علامات ترقيم بشكل صحيح حتى تُحفظ.

وتؤكد هذه الفرضية بحالات اختبار إضافية.

نصائح لإيجاد العيوب

حالما تتثبت من خطأ صفّي حالة الاختبار التي تنتجه، يصبح إيجاد مصدر الخطأ إما سهلاً أو فيه تحدٍّ، حسب مدى الجودة الذي كتبت به شفرتك. إذا كنت تكابد وقتاً صعباً في إيجاد عيب، فربما هذا بسبب أن شفرتك ليست مكتوبة بشكل جيد. قد لا ترغب ب سماع هذا، لكنها الحقيقة. إن كنت تواجه مشاكل، فكر بالنصائح التالية:

استخدم كل البيانات المتاحة لتصنع فرضيتك عندما تنشئ فرضية عن مصدر خطأ، احسب حساب قدر ما تستطيع من البيانات في فرضيتك. في المثال، قد تلاحظ أن *Fruit-Loop, Frita* كانت خارج الترتيب وتنشئ فرضية أن الأسماء التي تبدأ ب "F" تُرتّب بشكل غير صحيح. هذه فرضية فقيرة لأنها لا تحسب حساب حقيقة أن *Modula, Mildred* كانت خارج الترتيب أو أن الأسماء تُرتّب بشكل صحيح خلال المرة الثانية. إن لم تلائم البياناتُ الفرضية، لا تهمل البيانات-تساءل لمَ لا تلائمها، وأنشئ فرضية جديدة.

الفرضية الثانية في المثال-أن المشاكل تظهر من الأسماء التي تحوي شحطات، وليس من الأسماء التي تُدخل بشكل إفرادي-لم تبدو في البداية أنها تحسب حساب حقيقة أن الأسماء ترتّب بشكل صحيح في المرة الثانية أيضاً. في هذه الحالة، على كل، توصِلنا الفرضية الثانية إلى فرضية أكثر صفاء والتي أُثبت أنها صحيحة. صحيح تماماً ألا تحسب الفرضية حساب كل البيانات في البداية طالما أنك تتابع في تنقية الفرضية حتى تفعل في النهاية.

صفّ حالات الاختبار التي أنتجت الخطأ إن لم تستطع إيجاد مصدر الخطأ، حاول أن تصفي حالات الاختبار أكثر مما هي عليه. قد تكون قادراً على تغيير أحد الوسطاء أكثر مما افترضت، والتركيز على واحد من الوسطاء قد يمدّك بالفصل الحاسم.

نفذ الشفرة بعدّة اختبار الوحدة خاصتك[1] تميل العيوب لتكون أسهل إيجاداً في قطّاعات الشفرة الصغيرة بالمقارنة مع البرامج المدمجة الكبيرة. استخدم اختبارات الوحدة الخاصة بك لتختبر الشفرة بانعزال.

استخدم الأدوات المتاحة أدوات عديدة متاحة لتدعم جلسات *التصحيح*: *المصححات* التفاعلية، والمترجمات الانتقائية، وفاحصات الذاكرة، والمحررات الموجهة بالقواعد، وما إلى ذلك. تستطيع الأداة الصحيحة أن تجعل عملا صعباً سهلاً. مع خطأ من العسير إيجاده، على سبيل المثال، أحد أقسام البرنامج كان

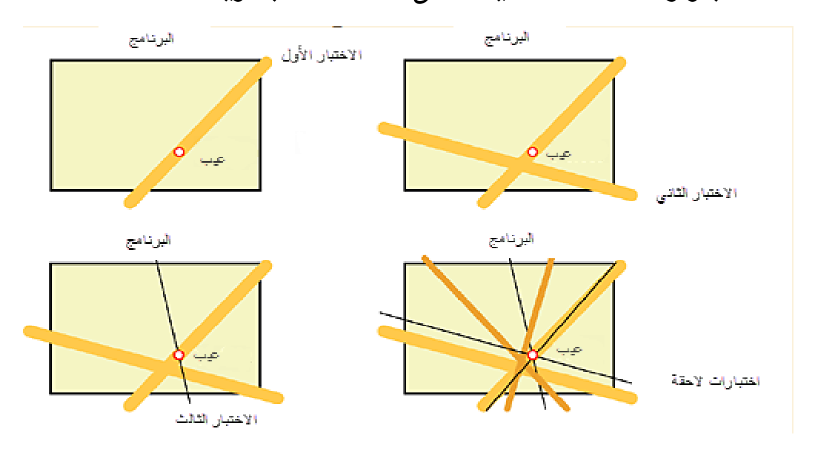
[1] **إشارة مرجعية** للمزيد عن منصات اختبار الوحدة، انظر "Plug unit tests into a test framework" في القسم 22. 4

يكتب فوق ذاكرة قسم آخر. هذا الخطأ من الصعب أن يُشخص باستخدام تطبيقات *التصحيح* المألوفة لأن المبرمج لا يستطيع أن يحدد النقطة المحددة التي عندها كان البرنامج يكتب كتب البرنامج فوق الذاكرة بشكل غير صحيح. استخدم المبرمج "نقطة توقف" ليضع مراقب على عنوان ذاكرة محدد. عندما كتب البرنامج على ذلك الموضع من الذاكرة، توقف *المصحح* وفُضحت الشفرة المجرمة.

هذا مثال عن مشكلة من الصعب أن تُشخص تحليلياً لكن هذا يصبح بسيط تماماً عندما تُطبق الأداة الصحيحة.

أعد إنتاج الخطاء بعدة طرق مختلفة أحياناً تجريب حالات مشابهة لحالة إنتاج الخطأ لكن ليست هي تماماً يكون هادياً. فكّر بهذا النهج كتثليث للعيب. إذا استطعت أن تحصل على إحداثياته من نقطة وعلى إحداثيات أخرى من نقطة أخرى، تستطيع بشكل أفضل أن تحدد بالضبط أين هو.

كما هو واضح في الشكل 23-1، إعادة إنتاج خطأ بعدة طرق مختلفة يساعد في تشخيص سبب الخطأ. حالما تعتقد أنك حددت العيب، شغّل حالة قريبة من الحالات التي أنتجت أخطاء لكن ينبغي ألا تنتج الخطأ نفسه. إذا أنتجت الخطأ بالتأكيد، فإنك لم تفهم المشكلة تماماً بعد. تظهر الأخطاء عادةً من تجميعات من العوامل، ومحاولة تشخيص المشكلة بحالة اختبار واحدة عادةً لا يشخص المشكلة الجذرية.



الشكل 23-1 حاول إنتاج الخطأ بعدة طرق مختلفة لتحدد سببه الدقيق.

ولّد بيانات أكثر لتولّد فرضيات أكثر اختر حالات اختبار مختلفة عن حالات الاختبار التي تعرفها مسبقاً لتكون إما خاطئة أو صحيحة. شغلّها لتولد بيانات أكثر، واستخدم البيانات الجديدة لتضيف الفرضية الممكنة إلى اللائحة.

استخدم نتائج الاختبارات السلبية لنفرض أنك أنشأت فرضية وأجريت حالة اختبار لتثبتها. لنفرض أكثر أن حالة الاختبار تلك نفت الفرضية، لذا لا تزال لا تعلم مصدر الخطأ. إنك تعلم شيء ما لم تكن تعلمه من قبل- أعني، أن العيب ليس في المكان الذي اعتقدت. هذا يضيّق ميدان بحثك ومجموعة الفرضيات الممكنة الباقية.

قم بعصف ذهني من أجل فرضيات ممكنة بدلاً من أن تلزم نفسك بأول فرضية فكرت بها، حاول أن تأتي بالعديد. لا تحللها في البداية-فقط آت بقدر ما تستطيع في عدة دقائق. ثم انظر إلى كل فرضية وفكر بحالات اختبار تثبتها أو تنفيها. هذا التمرين الذهني نافع في تكسير سدادة *التصحيح* والتي تنتج من التركيز بشكل جادً جداً على خط مفرد من الاستنتاج.

حافظ على دفتر ملاحظات على مكتبك، واصنع لائحة تغييرات للتجريب أحد أسباب استعصاء المبرمجين خلال جلسات *التصحيح* هو أنهم ينزلون بعيداً جداً في المسارات ذات النهاية المسدودة. اصنع لائحة أشياء للتجريب، وإن لم ينجح أحد النُهج، تقدم إلى النهج التالي.

ضيّق المنطقة المشبوهة من الشفرة إذا كنت تختبر برنامجاً كاملاً أو صفاً كاملاً أو إجرائية، اختبر أقسام أصغر بدلاً. استخدم عبارات طباعة، كتابة في سجل، أو اقتفاء أثر لتحدد أي قسم من الشفرة ينتج الخطأ.

إذا كنت تريد تقنية أقوى لتضييق المنطقة المشبوهة من الشفرة، أزل بشكل نظامي أجزاء من البرنامج وانظر إن كان الخطأ لايزال يحدث. إن لم يفعل، فإنك تعرف أنه في القسم الذي نزعته. إن فعل، فإنك تعرف أنه في القسم الذي أبقيته.

بدلاً من إزالة مناطق اعتباطياً، فرق تسد. استخدم خوارزمية البحث الثنائي لتركز بحثك. حاول أن تزيل حوالي نصف الشفرة في المرة الأولى. حدد النصف الحاوي على العيب، فقسّم هذا الجزء. مجدداً، حدد أي نصف يحتوي على العيب، ومجدداً، اقطع هذا الجزء من المنتصف. تابع حتى تجد العيب.

إن كنت تستخدم العديد من الإجرائيات الصغيرة، ستكون قادراً على إزالة أجزاء من الشفرة ببساطة عن طريق "تعليق" استدعاءات الإجرائيات. وإلا، تستطيع استخدام التعليقات أو أوامر المعالج التمهيدي لتزيل الشفرة.

إذا كنت تستخدم *مصححاً*، فإنك لست مضطراً لإزالة قطع من الشفرة. تستطيع أن تضع "نقاط توقف" على فرع من الطريق خلال البرنامج وتفحص العيب في ذلك الطريق بدلاً. إن سمح لك *مصححك* بتجاوز استدعاءات الإجرائيات، اعزل المشتبهين بتجاوز تنفيذ إجرائيات محددة والنظر إن كان الخطأ لا يزال يحدث. العملية مع *المصحح* مشابهة من وجه آخر للعملية التي بها تُزال قطع من البرنامج عضوياً.

كن شاكّاً بالصفوف والإجرائيات التي امتلكت عيوباً في الماضي[1] الصفوف التي امتلكت عيوباً من قبل مرجحة لمواصلة امتلاك العيوب. صف كان فيه مشاكل في الماضي أكثر عرضة ليحوي عيوب جديدة من صف كان خال من العيوب. أعد امتحان الصفوف والإجرائيات الميالة للخطأ.

افحص الشفرة التي تغيرت مؤخراً إن امتلكت خطأ جديداً صعب التشخيص، فإنه عادةً متعلق بشفرة غيرتها مؤخراً. يمكن أن يكون في شفرة جديدة بالكامل أو في تغييرات في شفرة قديمة. إن لم تستطع إيجاد عيب، شغل إصداراً قديماً من البرنامج لترى إن كان العيب يحدث. إن لم يكن، فإنك تعلم أن الخطأ في الإصدار الجديد أو ناتج عن تفاعل[2] في الإصدار الجديد. أمعن النظر في الفوارق بين الإصدارين. افحص سجل التحكم بالإصدار لترى أي شفرة تغيرت مؤخراً. إن لم يكن هذا ممكناً، استخدم أداة مقارنة لتقارن التغيرات بين الشفرة المصدرية الشغالة القديمة والشفرة المصدرية العاطلة الجديدة.

وسّع المنطقة المشبوهة من الشفرة من السهل أن تركز على جزء صغير من الشفرة، متأكداً أن "العيب يجب *حتماً* أن يكون في هذا القسم." إن لم تجده في الجزء، فكر باحتمالية أن العيب ليس في الجزء. وسع مساحة شكك، ثم ركز على قطع منها باستخدام تقنية البحث الثنائي الموصوفة سابقاً.

كامل بشكل تزايدي[3] يكون *التصحيح* سهلاً إن أضفت قطعاً على نظام واحدة في كل مرة. إذا أضفت قطعة إلى نظام وواجهت خطأ جديداً، أزل القطعة واختبرها بشكل منفصل.

ابحث عن العيوب الشائعة استخدم لوائح جودة الشفرة لتحفز تفكيرك حول العيوب المحتملة. إذا كنت تتبع تطبيقات التفحص الموصوفة في القسم 21. 3 "التفحصات الرسمية،" سيكون لديك لائحة الاختبار "حسنة المعايَرة" الخاصة بك عن المشاكل الشائعة في بيئتك. تستطيع أيضاً استخدام لوائح الاختبار التي تظهر في هذا الكتاب. انظر "لائحة لوائح اختبار" التالية لجدول محتويات الكتاب.

[1] **إشارة مرجعية** لتفاصيل أكثر حول الشفرة الميالة للخطأ، انظر "استهدف وحدات ميالة للخطأ" في القسم 24. 5

[2] (فعل الانتقال بين الشفرتين القديمة والجديدة)

[3] **إشارة مرجعية** لنقاش كامل حول التكامل، انظر الفصل 29، "التكامل"

تحدث إلى شخص آخر عن المشكلة[1] يدعو بعض الناس هذا ب "*تصحيح كرسي الاعتراف*." غالباً ما تكتشف عيبك الخاص عند القيام بشرحه إلى شخص آخر. مثالاُ، إن كنت تشرح المشكلة في مثال الرواتب، قد يبدو الامر كالآتي:

مرحبا، صديقي، هل أستطيع أخذ دقائق من وقتك؟ عندي مشكلة. حصلت على هذه اللائحة من رواتب الموظفين التي يُفترض أن تكون مرتبة، لكن بعض الأسماء خارجة عن الترتيب. إنها تترتب بشكل صحيح تماماً في المرة الثانية التي أطبعها لكن ليس من الأولى. لقد فحصت لأرى إن كانت المشكلة في الأسماء الجديدة، لكنني جربت وبعض المحاولات نجحت. أنا أعلم أنه ينبغي أن تُرتب من المرة الأولى التي أطبعها فيها لأن البرنامج يرتب كل الأسماء طالما تُدخل ويرتبها مجدداً عندما تُخزن-انتظر لحظة-لا لا، إنه لا يرتبها عندما تُدخل. هذا صحيح. إنه يرتبها بشكل تقريبي فقط. شكراً، صديقي. لقد قدمت مساعدة كبيرة.

لم ينطق الصديق بكلمة، وأنت الذي حللت مشكلتك. هذه النتيجة تحصل عادةً، وهذا النهج هو أداة فعالة لحل العيوب الصعبة.

خذ فاصل عن المشكلة أحياناً تركز بجد كثيراً حتى لا تعد تستطيع التفكير. كم مرة توقفت لشرب فنجان من القهوة وحللت المشكلة في طريقك إلى آلة القهوة؟ أو في منتصف الغداء؟ أو في طريقك إلى المنزل؟ أو في الدش "الحمام" في الصباح التالي؟ إذا كنت *تصحح* ولم تنجز أي تقدم، حالما تجرب كل الطرق، اجعل هذه الفترة فترة راحة. اذهب للمشي. اعمل في شيء ما مختلف. عد إلى المنزل في هذا اليوم. دع عقلك اللاواعي يحصل بحريته على حل للمشكلة.

الفائدة الإضافية من الاستسلام المؤقت هو أنه يخفّض القلق المرتبط *بالتصحيح*. بداية القلق هي إشارة واضحة على أنه وقت فاصل الراحة.

تصحيح القوة العمياء

القوة العمياء هي نهج غالباً ما يهمل في *تصحيح* مشاكل البرمجيات. أشير ب "القوة العمياء" إلى تقنية قد تكون مملة وشاقة ومستهلكة للوقت لكنها *تضمن* حل المشكلة. التقنيات المحددة التي *تضمن* حل المشكلة تكون تابعة للسياق، لكن هنا بعض المرشحين بالعموم:

- قم بمراجعة كل التصميم و|أو الشفرة في الشفرة العاطلة.
- ارم ذاك القسم من الشفرة بعيداً وأعد التصميم|كتابة الشفرة من الصفر.
- ارم كامل البرنامج بعيداً وأعد التصميم|كتابة الشفرة من الصفر.

---

[1] **إشارة مرجعية** لتفاصيل حول كيف يضع تضمين مطورين آخرين مسافة مفيدة بينك وبين المشاكل، انظر القسم 21. 1، "نظرة عامة على تطبيقات التطوير التعاوني."

- ترجم الشفرة مع معلومات التصحيح الكاملة.
- ترجم الشفرة في مستوى التحذير الأكثر انتقائية وأصلح تحذيرات المترجم الانتقائية هذه.
- جهز عدة اختبار الوحدة واختبر الشفرة الجديدة بمفردها.
- أنشئ عدة اختبار آلية وشغلها بشكل صحيح تماماً.
- اخطُ يدوياً فوق حلقة كبيرة في المصحح حتى تحصل على شرط الخطأ.
- سيّر شفرتك بعبارات الطباعة أو الإظهار أو أي تسجيل آخر.
- ترجم الشفرة بمترجم آخر.
- ترجم وشغل البرنامج في بيئة مختلفة
- اربط أو شغّل الشفرة بمكتبات أو بيئات تنفيذ خاصة تؤمن تحذيرات عندما تُستخدم الشفرة بشكل خاطئ
- اضبط على إعدادات الآلة الكاملة الخاصة بالمستخدم النهائي.
- ادمج الشفرة الجديدة بقطع صغيرة، تُفحص بشكل كامل حالما تُدمج.

عيّن الزمن الأعظمي للتصحيح السريع والمتسخ من أجل أي تقنية من تقنيات القوة العمياء، قد تكون ردة فعلك، "لا أستطيع فعل هذا-فيه الكثير من العمل!" النقطة هنا أن فيه الكثير من العمل إن أخذ زمناً أكثر مما أسميه "*التصحيح السريع والمتسخ*." دائماً ما يصيبك إغراء لتجرب تخميناً سريعاً بدلاً من قيادة أوركسترا الشفرة وحرمان العيب من أي مكان للاختباء. يفضّل المقامر داخل كل منا استخدام النهج الخطير الذي قد يجد العيب في خمس دقائق على النهج المضمون الذي سيجد العيب في نصف ساعة. الخطورة تكمن في أنك ستعْلَق، إن فشل نهج الدقائق الخمس. إذ يصبح إيجاد العيب بالطريقة السهلة قضية مبدئية، وتمر ساعات غير منتجة، كما تمر أيام وأسابيع وأشهر .... كم مرة قضيت ساعتين في *تصحيح* شفرة أخذت 30 دقيقة في الكتابة؟ هذا توزيع سيء للجهد، وستكون في حال أفضل إن أعدت كتابة الشفرة عما ستكون عليه إن *صححت* شفرة سيئة.

عندما تقرر الذهاب إلى نصر سريع، عيّن نهاية الزمن العليا لتجريب الطريقة السهلة. إذا تجاوزت نهاية الزمن، انسحب بنفسك إلى الفكرة التي تقول إن العيب سيكون أصعب تشخيصاً مما اعتقدت أصلاً، وأخرجه من مختبأه بالطريقة الصعبة. يسمح لك هذا النهج بطرد العيوب السهلة بعيداً على نحو صحيح والعيوب الصعبة بعد زمن أطول بالقليل.

اصنع لائحة بتقنيات القوة العمياء قبل أن تبدأ *التصحيح* لخطأ صعب؛ اسأل نفسك، "إن علقت وأنا *أصحح* هذه المشكلة، هل من طريقة أضمن بها أنني قادر على إصلاح المشكلة؟" إن استطعت تحديد تقنية قوة

عمياء واحدة على الأقل تستطيع حل مشكلتك-إعادة كتابة الشفرة في السؤال، داخلة ضمناً-فلن تكون إضاعتك لساعات أو أيام مرجحة عندما يوجد بديل أسرع.

الأخطاء القواعدية

تسير مشاكل الأخطاء القواعدية على طريق الماموث الغامض والنمر ذو الأسنان المدببة. تصبح المترجمات أفضل في إعطاء رسائل التشخيص، والأيام التي كان عليك فيها أن تقضي ساعتين في إيجاد فاصلة منقوطة موضوعة في مكان غلط في لائحة بلغة **باسكال** ولّت تقريباً. إليك لائحة بتوجيهات تستطيع اتباعها لتسرّع انقراض هذه الأصناف المعرضة للانقراض:

لا تثق بأرقام الأسطر في رسائل المترجم عندما يعطي مترجمك تقاريراً بخطأ قواعدي غامض، انظر مباشرة قبل ومباشرة بعد الخطأ-قد يكون المترجم قد أساء فهم المشكلة أو قد يمتلك ببساطة تشخيصات فقيرة. حالما تجد العيب الحقيقي، حاول أن تحدد لم وضع المترجم الرسالة على العبارة الغلط. الفهم الأفضل لمترجمك يمكن أن يساعدك بإيجاد عيوب مستقبلية.

لا تثق برسائل المترجم تحاول المترجمات إخبارك بالضبط ما هو الغلط، لكن المترجمات أوغاد صغيرة مخادعة، وغالبا يجب عليك أن تقرأ ما بين السطور لتعرف ما يعني أحدها بالفعل. مثالاً، في **سي يونكس**، يمكن أن تتلقى رسالة تقول "استثناء عوم" لعدد صحيح قُسّم على 0. مع مكتبة قوالب **سي** ++ المعيارية، يمكن أن تتلقى زوجاً من رسائل الخطأ: الأولى هي الخطأ الفعلي في استخدام المكتبة؛ الثانية رسالة من المترجم تقول، "رسالة خطأ طويل جداً على الطابعة لتطبعه؛ اقتُطِعت الرسالة." (“`Error message too long for printer to print; message truncated`.”) من الممكن أن تجد العديد من الأمثلة بنفسك.

لا تثق برسالة المترجم الثانية بعض المترجمات أفضل من بعض في اكتشاف أخطاء متعددة. تتحمس بعض المترجمات جداً بعد اكتشاف الخطأ الأول بحيث تصبح طائشة ومفرطة الثقة بنفسها، إنها تثرثر بدزينات من رسائل الخطأ التي لا تعني أي شيء. مترجمات أخرى أكثر رزانة، وعلى الرغم من الوجوب الحتمي لأن تشعر بإحساس الإنجاز عندما تكتشف خطأ، فإنها تمتنع عن تقيؤ رسائل غير دقيقة. عندما يولد مترجمك سلسلة رسائل خطأ متتالية، لا تقلق إن لم تستطع بسرعة أن تجد مصدر رسالة الخطأ الثانية أو الثالثة. أصلح الأولى وأعد الترجمة.

فرق تسد فكرة تقسيم البرنامج إلى أقسام للمساعدة في اكتشاف العيوب تعمل بشكل جيد بشكل خاص مع الأخطاء القواعدية. إن كان لديك خطأ قواعدي مثير للمشاكل، أزل قسماً من الشفرة وترجم مجدداً. إما أن لا

تحصل على خطأ (لأن الخطأ في القسم المزال)، أو تحصل على نفس الخطأ (يعني أنك تحتاج أن تزيل قسماً مختلفاً)، أو تحصل على خطأ مختلف (لأنك غششت المترجم حتى قدّم رسالة فيها معنى أكثر).

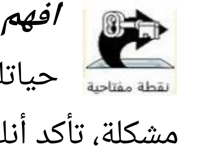
جد التعليقات وعلامات التنصيص الموضوعة في أماكن خاطئة[1] ينسق العديد من المحررات النصية البرمجية التعليقات والسلاسل المحرفية والعناصر القواعدية الأخرى بشكل آلي. في بيئات أكثر بدائية، يمكن لتعليق أو إشارة تنصيص موضوعين في مكان خاطئ أن يُفشّلا المترجم. لتجد التعليق أو علامة التنصيص الزائدين، ادخل السلسلة التالية إلى شفرتك في سي وسي++ وجافا:

```
/*"/**/
```

عبارة الشفرة هذه ستنهي إما تعليقاً أو سلسلة نصية، والذي يكون مفيداً في تضييق المسافة التي يختبئ فيها التعليق أو السلسلة النصية غير المنتهين

23.3 إصلاح عيب

القسم الصعب من التصحيح هو إيجاد العيب. إصلاح العيب هو القسم السهل. لكن كما مع العديد من المهام السهلة، حقيقة كونها سهلة يجعلها ميالة للخطأ. وجدت دراسة واحدة على الأقل أن تصحيح العيوب لديه فرصة أكثر من 50 بالمئة ليكون غلط من المرة الأولى (يوردون b1986). هنا بضعة توجيهات لتقليل الفرصة على الخطأ:



افهم المشكلة قبل أن تصلحها "دليل الشيطان في *التصحيح*" صحيح: أفضل طريقة لتجعل حياتك أصعب ولتَحُتَّ جودة برنامجك هي أن تصلح المشاكل بدون أن تفهمها حقاً. قبل أن تصلح مشكلة، تأكد أنك فهمتها إلى نواتها. ثلّث العيب بكلا الحالات التي ينبغي أن تعيد إنتاج الخطأ والحالات التي لا ينبغي أن تعيد إنتاج الخطأ. تابع على هذا حتى تفهم المشكلة بشكل جيد كفاية لتتوقع حدوثها بشكل صحيح في كل مرة.

افهم البرنامج، وليس المشكلة فقط إذا فهمت السياق الذي تحدث فيه المشكلة، فإنك مرجح أكثر لتحل المشكلة بشكل كامل وليس بفهم جانب واحد منها. وجدت دراسة على البرامج القصيرة أن لدى المبرمجين الذين يحققون فهماً شاملاً لسلوك البرنامج فرصة أفضل بتعديله بنجاح من المبرمجين الذين يركزون على السلوك الموضعي، ويتعلمون عن البرنامج فقط عندما يحتاجون لذلك (لقمان وآخرون. 1986). لأن البرنامج في هذه الدراسة كان صغيراً (280 سطراً)، فإن هذا لا يثبت أن عليك أن تحاول فهم برنامج ب 50000 سطر

[1] **إشارة مرجعية** إمكانية الحصول على محررات موجهة بالقواعد هي إحدى المميزات بين بيئات البرمجة في موجة الطفولة وموجة النضج. لتفاصيل، انظر القسم 4. 3، "موقعك على الموجة التكنولوجية."

بشكل كامل قبل أن تصلح العيب. إن هذا يقترح أن عليك أن تفهم على الأقل الشفرة بجوار التصحيح للعيب - "الجوار" لا يكون بضعة أسطر بل بضعة مئات.

أكّد تشخيص العيب قبل أن تندفع إلى إصلاح عيب، تأكد أنك شخصّت المشكلة بشكل صحيح. خذ وقتك لتجري حالات اختبار تثبت فرضيتك وتنفي الفرضيات المنافسة. إذا أثبتَّ فقط أن المشكلة يمكن أن تكون نتيجة لواحد من عدة أسباب، فإنك حتى الآن لا تملك البرهان لتعمل على ذلك السبب الواحد؛ أبطل الآخرين أولاً.

استرخِ[1] كان مبرمج جاهزاً لينطلق إلى رحلة تزلج. كان منتجه جاهزاً للتسويق، لقد كان متأخراً من قبل، وكلن لديه عيب واحد فقط للإصلاح. لقد غيّر الملف المصدري ووضعه في المتحكم بالإصدار. لم يعد ترجمة البرنامج ولم يتأكد أن التغيير كان صحيحاً.

بالحقيقة، التغيير كان غير صائب، وكان مديره ممتعضاً. كيف يمكن أن يغيّر الشفرة في منتج كان جاهزاً للتسويق بدون فحصه؟ ماذا يمكن أن يكون أسوأ؟ أليس هذا قمة التهور المهني؟

إن لم يكن هذا قمة التهور، فإنه قريب وهو شائع. الإسراع إلى حل مشكلة واحدٌ من الأبعد عن الفعالية الزمنية التي يمكن أن تفعلها. إنه يقود إلى محاكمة مستعجلة، وتشخيص غير مكتمل لعيب، وتصحيحات غير كاملة. يمكن أن يقودك التفكير الحريص إلى رؤية حلول حيث لا توجد الحلول. الضغط-غالباً المفروض ذاتياً- يحثّ على حلول تجربة-وخطأ تصادفية وعلى افتراض أن حلاً يعمل بنجاح بدون التأكد من أنه يفعل.

في الطرف المقابل البارز، خلال الأيام الأخيرة من تطوير **مايكروسوفت ويندوز 2000**، احتاج مطور أن يصلح عيباً كان الأخير الباقي قبل يُفسح المجال حتى تُنشأ **النسخة المرشحة للإطلاق**. غيّر المطور الشفرة، وفحص إصلاحه، واختبر إصلاحه على بناءه المحلي (`local build`). لكنه لم يضعه في التحكم بالإصدار في تلك اللحظة. بدلاً، ذهب للعب كرة السلة. قال: "أشعر أنني مضغوط جداً الآن حتى أتأكد أنني حسبت حساب كل شيء يجب أن أحسب حسابه. أنا ذاهب لأصفّي مخّي لساعة، وبعدها سأعود وسأضع الشفرة (في التحكم بالإصدار) -حالما أقنع نفسي أن ذلك الإصلاح كان صحيحاً بالفعل."

استرخِ لفترة طويلة كفاية لتتأكد أن حلك صحيح. لا تُفتتن بسلوك الطرق المختصرة. قد تأخذ وقتاً أطول، لكنها ربما تأخذ أقل. إن لم يكن هنالك شيء آخر، ستصلح المشكلة بشكل صحيح ولن يرجعك مديرك من رحلة التزلج.

[1] لا تصحح أبداً وانت واقف. --جيرالد وينبرغ

احفظ الشفرة المصدرية الأصلية[1] قبل أن تبدأ بإصلاح العيب، تأكد أن تؤرشف إصداراً من الشفرة تستطيع الرجوع إليه لاحقاً. من السهل أن تنسى أي تغيير في مجموعة التغييرات هو التغيير المُعتبر. إن كان لديك الشفرة المصدرية الأصلية، على الأقل تستطيع أن تقارن الملفات القديمة والجديدة وترى أين توجد التغييرات.

أصلح المشكلة، لا العرض ينبغي أن تصلح العرض أيضاً، لكن ينبغي أن يكون التركيز على إصلاح المشكلة الجذر بدلاً من تغليفها ببكرة قنوات برمجية. إن لم تفهم المشكلة بعمق، فإنك لا تصلح الشفرة. إنك تصلح العرض وتجعل الشفرة أسوأ. افترض أنه لديك الشفرة التالية:

مثال جافٍ عن شفرة تحتاج إلى إصلاح

```
for ( claimNumber = 0; claimNumber < numClaims[
     client ]; claimNumber++ ) {
  sum[ client ] = sum[ client ] + claimAmount[
                     claimNumber ];
}
```

افترض أكثر أنه عندما يساوي *client 45*، يصبح *sum* غلطاً بخطأ 3.45. إليك الطريقة الغلط لإصلاح المشكلة:

شفرة مرعبة

مثال جافٍ عن جعل الشفرة أسوأ أ"ب"إصلاحها"

```
for ( claimNumber = 0; claimNumber < numClaims[
     client ]; claimNumber++ ) {
  sum[ client ] = sum[ client ] + claimAmount[
                     claimNumber ];
}
if ( client == 45 ) {
  sum[ 45 ] = sum[ 45 ] + 3.45;
}
```

هذا "الإصلاح"

الآن افترض أنه عندما يساوي *client 37* وعدد المطالبات للزبون *0*، فإنك لا تحصل على *0*. إليك الطريقة الغلط بإصلاح المشكلة:

---

[1] **إشارة مرجعية** نوقشت القضايا العامة المتضمنة في تغيير الشفرة بعمق في الفصل 24، "إعادة التصنيع"

```
مثال جافا عن جعل الشفرة أسوأ ب"إصلاحها"، تتمة
for ( claimNumber = 0; claimNumber < numClaims[
     client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[
                  claimNumber ];
}
if ( client == 45 ) {
   sum[ 45 ] = sum[ 45 ] + 3.45;
}
else if ( ( client == 37 ) && ( numClaims[ client ]
                                 == 0 ) ) {
   sum[ 37 ] = 0.0;
}
```

شفرة مرعبة

هذا "الإصلاح" الثاني

إن لم يبعث هذا قشعريرة باردة أسفل عمودك الفقري، فلن تتأثر بأي شيء آخر في هذا الكتاب أيضاً. من المستحيل سرد كل المشاكل في هذا النهج في كتاب بحوالي 1000 صفحة فقط، لكن إليك الثلاثة في القمة:

- لن تعمل الإصلاحات في معظم الأحيان. تبدو المشاكل وكأنما هي نتيجة لعيوب التهيئة. عيوب التهيئة هي، حسب التعريف، غير قابلة للتوقع، لذا حقيقة أن المجموع للزبون 45 هي بعيدة ب 3.45$ اليوم لا تخبرك أي شيء عن الغد. فد تكون بعيدة ب 10000.02$، أو قد تكون صحيحة. هذه هي طبيعة عيوب التهيئة.
- إنها غير قابلة للصيانة. عندما تكون الشفرة مخصصة لحالات للإفلات من الأخطاء، تصبح الحالات الخا صة الميزة الأبرز في الشفرة. ال 3.45$ لن تكون دائماً 3.45$، و سيظهر خطأ آخر لاحقاً. ستُعدل الشفرة مجدداً لتتعامل مع الحالة الخاصة الجديدة، ولن تُزال الحالة الخاصة ل 3.45$. ستصبح الشفرة بشكل متزايد مُلتصَقة ب الحالات الخا صة. أخيراً اللوا صق ستصبح ثقيلة جداً على الشفرة لتحملها، وستغرق الشفرة إلى أعماق المحيط-مكان مناسب لها.
- إنها تستخدم الحاسوب لشيء ما من الأفضل القيام به يدوياً. الحواسيب جيدة في الحسابات القابلة للتوقع والنظامية، لكن البشر أفضل في مراوغة البيانات بدهاء. ستكون حكيماً إذا عالجت الخرج بقلم تصحيح وآلة كاتبة بدلاً من قرد معه الشفرة.

غيّر الشفرة لسبب جيد فقط تقنية تغيير الشفرة بشكل عشوائي حتى تبدو أنها تعمل متعلقة بإصلاح الأعراض. يسير الخط النمطي للمنطقية كالتالي: "يبدو أنّ في هذه الحلقة عيب. من الممكن أن يكون خطأ "خارجاً من واحدة،" لذا سأضع *-1* هنا وأجرب. حسناً. لم تنجح، لذا فقط سأضع *+1* هنا بدلاً. حسناً. تبدو أنها تعمل. سأقول إنها أصلحت."

بمقدار انتشار هذا التطبيق، هو غير فعال. القيام بتغييرات على الشفرة بشكل عشوائي هو مثل تدوير إطارات **بونتياك أزتيك** لإصلاح مشكلة في المحرك. إنك لا تتعلم أي شيء، إنك فقط تعبث هنا وهناك. بتغيير البرنامج بشكل عشوائي، فإنك تقول فعلياً، "لا أعرف ماذا يجري هنا، لكن سأحاول بهذا التغيير وآمل أن ينجح." لا تغير

الشفرة بشكل عشوائي. إنها شعوذة برمجية. كلما جعلتها أكثر اختلافاً بدون فهم، قلّت الفرصة الثقة بأنها ستعمل بشكل صحيح.

قبل أن تقوم بتغيير، كن واثقاً أنه سينجح. كونك مخطئاً بتغيير ينبغي أن يتركك مدهوشاً. وينبغي أن يسبب الشك الذاتي، وإعادة تقييم النفس، وبحث روحي عميق. لذا فإنه ينبغي أن يحدث بندرة.

قم بتغيير واحد في المرة الواحدة التغييرات مخادعة كفاية عندما تُقام واحداً في المرة الواحدة. عندما يُقام اثنين في المرة الواحدة، يمكن أن تنتج أخطاء دقيقة تبدو مثل الأخطاء الأصلية. عندها تكون في موقف محرج من عدم معرفتك إن كنت لم تصحح الخطأ، أو إن صححت الخطأ لكن أنتجت واحداً جديداً يبدو مشابهاً، أو إن لم تصحح الخطأ وأنتجت واحداً جديداً مشابهاً. اجعلها بسيطة: قم بتغيير واحد فقط في المرة الواحدة.

اختبر إصلاحك[1] اختبر البرنامج بنفسك، أو دع أحداً ما يقوم بذلك لأجلك، أو اعبره مع شخص آخر. شغّل نفس حالات الاختبار التثليثية التي استخدمتها لتشخص المشكلة لتتأكد أن كل جوانب المشكلة قد انحلت. إن كنت قد حللت جزءاً فقط من المشكلة، فإنك ستكتشف أن لازال أمامك عمل لتقوم به.

أعد تشغيل البرنامج لتفحص الآثار الجانبية لتغييراتك. الطريقة الأسهل والأكثر فعالية لتفحص الآثار الجانبية هي أن تشغل البرنامج عبر طقم آلي لفحص التراجع في `Junit` أو `CppUnit` او شيء معادل.

أضف اختبار وحدة يكشف الأخطاء عندما تواجه خطأ لم يُكتشف بعدة الفحص خاصتك، أضف حالة اختبار لتكشف الخطأ بحيث لا يُعاد تقديمه لاحقاً.

ابحث عن العيوب المشابهة عندما تجد عيباً، ابحث عن العيوب الأخرى المشابهة. تميل العيوب لتحدث في مجموعات، وإحدى قيم لفت الانتباه إلى أنواع من العيوب تقوم أنت بصنعها هي أنك تستطيع تصحيح كل العيوب من ذلك النوع. البحث عن العيوب المشابهة يتطلب منك أن تمتلك فهماً عميقاً للمشكلة. انتبه إلى علامات التحذير: إن لم تستطع معرفة كيفية البحث عن عيوب مشابهة، فهذه إشارة على أنك لم تفهم المشكلة بشكل كامل بعد.

23. 4 الاعتبارات النفسية في التصحيح

التصحيح في مستوى الإلحاح الفكري لبقية نشاطات تطوير البرمجيات. تخبرك "أناك" نفسك أن شفرتك جيدة ولا تحتوي عيباً حتى عندما ترى أنها تحويه بالفعل. عليك أن تفكر بدقة-مشكلاً فرضيات، وجامعاً للبيانات،

[1] **إشارة مرجعية** لتفاصيل حول اختبار التراجع الآلي، انظر "إعادة الاختبار (اختبار التراجع)" في القسم 22. 6

ومحللاً للفرضيات، ورافضاً لها بمنهجية-مع شكلانية غير مألوفة لبعض الناس. إن كنت تبني الشفرة *وتصححها* معاً، يجب أن تتحول بسرعة بين التفكير الخلاق الرشيق الذي يسير مع التصميم والتفكير الحاسم بصلابة الذي يسير مع *التصحيح*. خلال قراءتك لشفرتك، عليك أن تصارع إلفة الشفرة وتدافع ضد رؤية ما تتوقع أن تراه.

كيف يساهم "الطقم النفسي" في عمى *التصحيح*

عندما ترى رمزاً في برنامج يقول *Num*، فماذا ترى؟ هل ترى غلط تهجئة للكلمة "Numb"؟ أو هل ترى اختصاراً ل"Number"؟ من شبه المؤكد أنك ترى اختصاراً ل"Number." هذه هي ظاهرة من ظواهر "الطقم النفسي"- ترى ما تتوقع أن تراه. ماذا تقول هذه اللافتة؟



في هذا اللغز التقليدي، يرى الناس غالباً "the" واحدة. يرى الناس ما يتوقعون رؤيته. اعتبر بالتالي:

- يتوقع غالباً الطلاب الذين يتعلمون حلقة *while* أن تكون الحلقة مقيَّمة بشكل مستمر؛ هذا يعني، إنهم يتوقعون من الحلقة أن تنتهي حالما يصبح شرط *while* خاطئاً، بدلاً من أن تكون مقيَّمة فقط في الأعلى أو في الأسفل (كرتيس وآخرون 1986). إنهم يتوقعون أن تتصرف حلقة *while* كما تفعل "While" في اللغة الإنكليزية.
- مبرمج استخدم بشكل غير مقصود كلا المتحولين *SYSTSTS* و*SYSSTSTS* اعتقد أنه كان يستخدم متحولاً وحيداً. لم يكتشف المشكلة حتى شُغّل البرنامج مئات المرات وكُتب كتاب حاوٍ على النتائج الخاطئة (وينبرغ 1998).
- مبرمج يلقي نظرة على شفرة مثل هذه الشفرة:

```
if ( x < y )
   swap = x;
   x = y;
   y = swap;
```

أحياناً يرى شفرة مثل هذه الشفرة:

```
if ( x < y ) {
   swap = x;
   x = y;
   y = swap;
}
```

يتوقع الناس من ظاهرة جديدة أن تماثل ظاهرة مشابهة رأوها من قبل. يتوقعون من تركيب التحكم الجديد أن يعمل مثل التراكيب القديمة؛ عبارات *while* في لفات البرمجة أن تعمل مثل عبارات "while" في الحياة الفعلية؛ وأن تكون أسماء المتحولات مثلما كانت سابقاً. إنك ترى ما تتوقع أن تراه وهكذا تغفل عن الفوارق، مثل تهجئتك الخاطئة لكلمة "لغات" في الجملة السابقة.

ماذا على الطقم النفسي أن يفعل مع *التصحيح*؟ أولاً، إنه يتكلم عن أهمية التطبيقات البرمجية الجيدة. يساعد التنسيق، والتعليق، وأسماء المتحولات، وأسماء الإجرائيات، الجيدة والعناصر الجيدة الأخرى من أسلوب البرمجة بتركيب الخلفيّة البرمجية بحيث تظهر العيوب المرجحة كانحرافات عن العيوب الأصلية وتدافع عن نفسها.

التأثير الثاني للطقم النفسي هو في اختيار أجزاء من البرنامج للفحص عندما يُكتشف خطأ. أظهر بحث أن المبرمجين الذين *يصححون* بالشكل الأكثر فعالية يشرّحون فكرياً أجزاء من البرنامج ليس لها صلة بالموضوع ويستبعدونها خلال *التصحيح* (باسيلي، وسيلبي، وهاتشن 1986). بالعموم، يسمح التمرين للمبرمجين الممتازين بتضييق حقول بحثهم وإيجاد عيوب بسرعة أكبر. أحياناً، على كل، القسم من البرنامج الذي يحوي العيب يُقتطع بشكل خاطئ ويستبعد. إنك تصرف وقتاً بتطهير قسم من الشفرة من العيب، وتتجاهل القسم الذي يحوي العيب. تأخذ منعطفاً خاطئاً في مفترق طرق وتحتاج إلى التراجع قبل أن تستطيع التقدم مجدداً. بعض الاقتراحات في نقاشات القسم 23. 2 عن اكتشاف العيوب مصمَّمة لقهر "عمى *التصحيح*" هذا.

كيف يمكن "للمسافة النفسية" أن تساعد

يمكن أن تُعرَّف المسافة النفسية على أنها السهولة التي بها يمكن أن يُميَّز عنصرين عن بعضهما.[1] إن كنت تبحث في لائحة طويلة من الكلمات وكنت قد أُخبرت أنها كلها عن البط، من الممكن بسهولة أن تظن أن "Queck" هي "Quack" لأن الكلمتين تبدوان نفس الشيء. المسافة النفسية بين الكلمتين صغيرة. سيكون لديك احتمال أقل أن تظن أن "Tuack" هي "Quack" حتى وإن كان الفرق حرف واحد مجدداً. "Tuack" هي أقل شبهاً ب "Quack" مما هي "Queck" لأن الحرف الأول في الكلمة أكثر بروزاً من واحد في المنتصف.

يسرد الجدول 23-1 أمثلة عن المسافة النفسية بين أسماء المتحولات

[1] **إشارة مرجعية** لتفاصيل حول تركيب أسماء متحولات ليست مربكة، انظر القسم 11. 7، "أنواع من الأسماء للتجنب."

جدول 23-1 **أمثلة عن المسافة النفسية بين أسماء المتحولات**

المتحول الأول	المتحول الثاني	المسافة النفسية
stoppt	stcppt	تقريباً غير مرئية
shiftrn	shiftrm	تقريباً لا توجد
dcount	bcount	صغيرة
claims1	claims2	صغيرة
product	sum	كبيرة

أنت *تصحح*، كن جاهزاً للمشاكل المسببة بالمسافة النفسية غير الكافية بين أسماء المتحولات المتشابهة وبين أسماء الإجرائيات المتشابهة. وأنت تبني الشفرة، اختر أسماء بفوارق أكبر بحيث تتجنب هذه المشاكل.

23.5 أدوات التصحيح—الواضحة والواضحة تقريباً

تستطيع القيام بالكثير من أعمال *التصحيح* التفصيلية والمرهقة للدماغ بأدوات *تصحيح* متوفرة بيسر.[1] الأداة التي ستغرّز الوتد الأخير في قلب العيب مصاص الدماء ليست متاحة بعد. لكن كل سنة تأتي بتحسينات متزايدة في المقدرات المتاحة.

مقارِنات الشفرة المصدرية

مقارِنات الشفرة المصدرية مثل `Diff` مفيدة عندما تعدّل برنامجاً كاستجابة للأخطاء. إن قمت بالعديد من التغييرات وتحتاج أن تلغي بعضاً ولا تتذكرها تماماً، يستطيع مقارن أن يحدد بدقة الفوارق وينبه ذاكرتك. إن اكتشفت عيباً في إصدار جديد لا تتذكره في إصدار أقدم، تستطيع أن تقارن الملفين لترى ما الذي تغير.

رسائل المترجم التحذيرية

إحدى أبسط أدوات *التصحيح* وأكثرها فعالية في مترجمك الخاص.

نقطة مفتاحية

عيّن مستوى التحذير في مترجمك إلى المستوى الأعلى والأكثر انتقائية الممكن، وأصلح الأخطاء التي يدوّنها من الفوضى تجاهل أخطاء المترجم. وأكثر وساخة أن تطفئ التحذيرات بحيث حتى لا تستطيع حتى أن تراها. يعتقد الأطفال أحياناً أنهم إن أغلقوا عيونهم ولم يستطيعوا رؤيتك، فإنهم جعلوك

[1] **إشارة مرجعية** الخط الفاصل بين الاختبار *والتصحيح* غامض. انظر القسم 22. 5 للمزيد عن أدوات الاختبار والفصل 30 للمزيد عن أدوات تطوير البرمجيات.

تذهب بعيداً. تعيين مفتاح في المترجم على وضع إطفاء التحذيرات يعني أنك لا تستطيع رؤية الأخطاء. إنه لا يجعلها تنقلع أكثر مما تجعل العيون المغمضة الكبار ينقلعون.

قدّر أن الناس الذين كتبوا المترجم يعرفون كمية عظيمة أكثر منك عن لغتك. إن كانوا يحذروك بخصوص شيء ما، فهذا يعني غالباً أنك تمتلك فرصة لتتعلم شيئا جديداً عن لغتك. ابذل الجهد لتفهم ما يعني التحذير بالفعل.

عامل التحذيرات كالأخطاء بعض المترجمات تدعك تعامل التحذيرات كالأخطاء. أحد أسباب استخدام هذه الميزة هو أنها ترفع الأهمية الظاهرة للتحذيرات. فقط كما يحتال تقديم ساعتك خمس دقائق عليك لتظن أنها بخمس دقائق بعد ما هي عليه، تعيين المترجم ليتعامل مع التحذيرات كأخطاء يحتال عليك حتى تتعامل معها بجدية أكثر. سبب آخر لتتعامل مع التحذيرات كالأخطاء هو أنها غالباً تؤثر بكيفية ترجمة برنامجك. عندما تترجم وتربط برنامجاً، فلن توقف التحذيرات بالحالة العادية البرنامج من الربط، لكن الأخطاء بالحالة العادية ستفعل. إن كنت تريد فحص التحذيرات قبل الربط، عين مفتاح المترجم على وضعية التعامل مع التحذيرات كالأخطاء.

هيّئ معايير على مستوى المشروع من أجل إعدادات وقت الترجمة عيّن معياراً يتطلب من كل شخص في الفريق أن يترجم شفرته باستخدام نفس إعدادات المترجم. وإلا، عندما تحاول أن تدمج الشفرات المترجمة من أشخاص مختلفين بإعدادات مختلفة، ستحصل على طوفان من رسائل الخطأ وكابوس دمج. هذا البند سهل التطبيق إذا استخدمت ملف صناعة أو تدوينة بناء معياريين في المشروع.

فحص القواعد والمنطق الموسَّع

تستطيع استخدام أدوات إضافية لفحص شفرتك بشكل أعمق مما يفعل مترجمك. مثالاُ، من أجل المبرمجين بلغة سي، أداة `lint` تبحث بعناية عن استخدام المتحولات غير المهيأة (كتابة = عندما تريد ==) والمشاكل الدقيقة بشكل مشابه.

موَصِّفات التنفيذ

قد لا تفكر بموصفات التنفيذ كأداة *تصحيح*، لكن عدة دقائق تُصرف في دراسة موصّف البرنامج يمكن أن تكشف الغطاء عن بعض العيوب المفاجئة (والمختبئة).

على سيبل المثال، اشتبهت بإجرائية إدارة ذاكرة في أحد برامجي على أنها كانت عنق زجاجة للأداء. كانت إدارة الذاكرة بالأصل مُكوِّن صغير يستخدم مصفوفة مرتبة خطيّاً من المؤشرات إلى الذاكرة. استبدلت جدول بعثرة (`hash table`) بالمصفوفة المرتبة خطياً بتوقع أن يقلّ زمن التنفيذ على الأقل إلى النصف. لكن بعد توصيف الشفرة، لم أجد تغيراً في الأداء بالمرة. قمت بفحص الشفرة بشكل أقرب ووجدت عيباً كان يضيّع مقداراً عظيماً من الوقت في خوارزمية التحصيص. لم يكن عنق الزجاجة تقنية البحث الخطي؛ لقد كان العيبَ لم أحتج أن

أحسّن البحث بعد كل ذلك. افحص خرج موصف التنفيذ لتدخل السرور إلى نفسك بأن برنامجك يصرف مقداراً معقولاً من الوقت في كل منطقة.

منصات\سقالات الاختبار

كما ذكرت في القسم 23. 2 في اكتشاف العيوب[1]، سحب قطعة ذات مشاكل من الشفرة خارجاً، وكتابة شفرة لاختبارها، وتنفيذها بمفردها هي الطريقة الأكثر فعالية في طرد الأرواح الشريرة من برنامج ميال للخطأ.

المصححات

تقدمت *المصححات* المتاحة تجارياً بثبات عبر السنين، والإمكانيات المتاحة اليوم يمكن أن تغيّر الطريقة التي تبرمج بها. تسمح لك *المصححات* الجيدة بوضع نقاط توقف للتوقف عندما يصل التنفيذ إلى سطر محدد، أو في المرة ن التي يصل فيها إليه، أو عندما يتغير متحول شامل، أو عندما تُسند قيمة محددة إلى متحول. إنها تسمح لك بالخطو عبر الشفرة سطراً بسطر، الخطو خلال أو فوق الإجرائيات. إنها تسمح للبرنامج بان ينفذ رجوعاً، والخطو خلفاً إلى النقطة التي عندها بدأ العيب. إنها تسمح لك بكتابة سجل لتنفيذ عبارات محددة—شيء مشابه لبعثرة عبارة الطباعة "أنا هنا!" في أنحاء البرنامج.

تسمح لك *المصححات* الجيدة بفحص كامل للبيانات، متضمنة التركيبات والبيانات المخصصة ديناميكياً. إنها تجعل من السهل رؤية محتوى لائحة مترابطة من المؤشرات أو مصفوفة مخصصة ديناميكياً. إنها لبيبة بأنماط البيانات المعرفة من قبل المستخدم. إنها تسمح لك بأن تصنع استعلامات ارتجالية عن البيانات، وأن تسند قيماً جديدة، وأن تتابع تنفيذ البرنامج.

تستطيع أن تنظر إلى اللغة العالية المستوى أو لغة التجميع المولدة من قبل مترجمك. إن كنت تستخدم لغات متعددة، فإن *المصحح* يعرض بشكل آلي اللغة الصحيحة لكل قسم من الشفرة. تستطيع أن تنظر إلى سلسلة استدعاءات لإجرائيات وبسرعة أن تعرض الشفرة المصدرية لأي إجرائية. تستطيع أن تغيّر محددات البرنامج في بيئة *المصحح*.

يتذكر الأفضل في *مصححات* اليوم أيضاً محددات *التصحيح* (نقاط التوقف، والمتحولات المراقَبة، وما إلى ذلك) لكل برنامج بمفرده بحيث لا يتوجب عليك أن تعيد إنشاءها من أجل كل برنامج *تصححه*.

[1] **إشارة مرجعية** لتفاصيل عن السقالات، انظر "بناء سقالة لاختبار الصفوف الفردية" في القسم 22. 5

تعمل *مصححات* النظام على مستوى الأنظمة بدلاً من مستوى التطبيقات لذا فإنها لا تتداخل مع تنفيذ البرنامج قيد *التصحيح*. إنها أساسية عندما *تصحح* برامجاً حساسة للتوقيت أو لمقدار الذاكرة المتاحة.

بإعطائك القوى الهائلة المقدّمة من قبل *المصححات* الحديثة، قد تتفاجأ إن انتقدها أحدهم. لكن بعضٌ من الناس الأكثر احتراماً في مجال علم الحاسوب ينصح بألا *تستخدمها*[1]. إنهم ينصحون باستخدام دماغك وتجنب أدوات *التصحيح* بالكامل. حجتهم هي أن أدوات *التصحيح* هي عكاز وإنك تجد المشاكل أسرع وبدقة أكبر بالتفكير بها بالمقارنة مع الاعتماد على الأدوات. إنهم يناقشون إنك أنت، وليس *المصحح*، من ينبغي عليه تنفيذ البرنامج فكرياً ليفرغه من العيوب.

بغض النظر عن البينة التجريبية، الحجة الأساسية ضد *المصححات* ليست صالحة. حقيقة إمكان إساءة استخدام أداة لا تقتضي وجوب رفضها. أنت لا تتجنب أخذ الأسبيرين نادراً لأنه من الممكن أن تزيد الجرعة. أنت لا تتجنب جز مرجتك الخضراء بجزازة عشب قوية فقط لأنه من المحتمل أن تجرح نفس

ك. أي أداة قوية يمكن أن تستخدم أو يساء استخدامها، وكذلك *المصححات*.



المصححات ليست بديلاً عن التفكير الجيد. لكن، في بعض الحالات، التفكير ليس بديلاً عن *المصححات* الجيدة أيضاً. الخليط الأكثر فعالية هو تفكير جيد ومصحح جيد.

لوائح اختبار: مذكرات التصحيح[2]

تقنيات لإيجاد العيوب

- استخدم كل البيانات المتاحة لتصنع فرضيتك.
- نقِّ حالات الاختبار التي تنتج الخطأ.
- مرن الشفرة بعدة اختبار الوحدة خاصتك.
- استخدم الأدوات المتاحة.
- أعد إنتاج الخطأ بطرق متعددة مختلفة.
- ولّد بيانات أكثر لتولد فرضيات أكثر.
- استخدم نتائج الاختبارات السلبية.
- قم بعصف ذهني للفرضيات المحتملة.
- احتفظ بمفكرة على مكتبك، واصنع لائحة أشياء للتجريب.

[1] *المصحح* التفاعلي هو مثال شامخ على ما لا يلزم—إنه يشجع على قرصنة التجربة والخطأ بدلاً من التصميم النظامي، وهو ايضاً يخبّئ الناس الهامشيين الذين هم بالكاد مؤهلون للبرمجة الدقيقة.

--هارلان ميلز

[2] cc2e.com/2368

- ضيّق المنطقة المشبوهة من الشفرة.
- اشتبه بالصفوف والإجرائيات التي كان فيها عيوب من قبل.
- افحص الشفرة التي تغيرت مؤخراً.
- وسّع المنطقة المشبوهة من الشفرة.
- كامل بشكل تزايدي.
- افحص بحثاً عن العيوب الشائعة.
- تكلم إلى شخص آخر عن المشكلة.
- خذ استراحة عن المشكلة.
- عيّن وقتاً أعظمياً للتصحيح السريع والوسخ.
- اصنع لائحة بتقنيات القوة العمياء، واستخدمها.

تقنيات للأخطاء القواعدية

- لا تثق بأرقام الأسطر في رسائل المترجم.
- لا تثق برسائل المترجم.
- لا تثق برسالة المترجم الثانية.
- فرق تسد.
- استخدم محرراً موجهاً بالقواعد لتجد التعليقات وعلامات التنصيص الموجودة في مكان خاطئ.

تقنيات لإصلاح العيوب

- افهم المشكلة قبل أن تصلحها.
- افهم البرنامج، لا المشكلة فقط.
- تأكد من تشخيص العيب.
- استرخ.
- احفظ الشفرة المصدرية الأصلية.
- أصلح المشكلة، لا العرض.
- غير الشفرة فقط لسبب جيد.
- قم بتغيير واحد في المرة الواحدة.
- افحص إصلاحك.
- أضف اختبار وحدة يكشف العيب.
- ابحث عن عيوب مشابهة.

النهج العام في التصحيح

- هل تستخدم التصحيح كفرصة لتتعلم أكثر عن برنامجك وأخطائك وجودة شفرتك ونهج حل المشاكل؟
- هل تتجنب نهج التجربة والخطأ المؤمِن بالخرافات في التصحيح؟
- هل تفترض أن الأخطاء من صنعك؟
- هل تستخدم الطريقة العلمية لتتثبت من الأخطاء المتقطعة (التي تظهر أحياناً وتختفي أحياناً)؟
- هل تستخدم الطريقة العلمية لتجد العيوب؟
- بدلاً من استخدام نفس النهج في كل مرة، هل تستخدم تقنيات متعددة مختلفة لتكتشف العيوب؟
- هل تتأكد من أن الإصلاح صحيح؟
- هل تستخدم رسائل المترجم التحذيرية، وموصِّفات التنفيذ، ومنصات الاختبار، والسقالات، والتصحيح التفاعلي؟

مصادر إضافية

المصادر التالية أيضاً تناقش *التصحيح*:

Agans, David J. Debugging: *The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. Amacom, 2003. يقدم هذا الكتاب مبادئ *التصحيح* العامة التي يمكن أن تُطبّق في لغة أو بيئة.

Myers, Glenford J. *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979. الفصل 7 من هذا الكتاب التراثي مكرس *للتصحيح*.

Allen, Eric. *Bug Patterns In Java*. Berkeley, CA: Apress, 2002. يرسم هذا الكتاب إطاراً لنهج *تصحيح* برامج **جافا** والذي هو مشابه جداً بالمفهوم لما شرح في هذا الفصل، متضمناً "الطريقة العلمية في *التصحيح*." ويفرق بين *التصحيح* والاختبار، ويحدد نماذج الثغرات الشائعة.

الكتابان التاليان متشابهان بأن عنوانيهما يوحيان أنهما قابلان للتطبيق فقط في برامج مايكروسوفت ويندوز و.NET، لكنهما يحتويان على نقاشات عن *التصحيح* بالعموم، ويستخدمان التأكيدات، وتطبيقات كتابة الشفرة التي تساعد على تجنب الثغرات في المقام الأول:

Robbins, John. Debugging Applications for Microsoft .NET and Microsoft Windows. Redmond, WA: Microsoft Press, 2003.

McKay, Everett N. and Mike Woodring. Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers. Boston, MA: Addison-Wesley, 2000.

نقاط مفتاحية

- التصحيح هو جانب من تطوير البرمجيات والذي يعطي نجاحاً باهراً أو فشلاً ذريعاً. النهج الأفضل هو أن تستخدم التقنيات الأخرى الموصوفة في هذا الكتاب لتتجنب العيوب في المقام الأول. ولا يزال يغني وقتك أن تحسّن من مهاراتك في *التصحيح*، بكل الأحوال، لأن الفرق بين أداء *التصحيح* الجيد والفقير هو على الأقل 10 إلى 1.
- النهج الخاضع للنظام في إيجاد وإصلاح الأخطاء هو عنصر حاسم للنجاح. ركز *تصحيحك* بحيث ينقلك كل اختبار خطوة إلى الأمام. استخدم **الطريقة العلمية في *التصحيح***.
- افهم المشكلة الجذر قبل أن تصلح البرنامج، التخمينات العشوائية عن مصادر الأخطاء والتصحيحات العشوائية ستترك البرنامج في حال أسوأ مما يكون عليه عندما تبدأ.
- ضع تحذيرات مترجمك على المستوى الأكثر انتقائية الممكن، وأصلح الأخطاء التي يدونها. من الصعب أن تصلح الأخطاء الماكرة إن تجاهلت الواضحة.
- أدوات *التصحيح* مساعدات قوية في تطوير البرمجيات. جدها واستخدمها، وتذكر أن تستخدم دماغك في نفس الوقت.

التصحيح

الفصل 24

إعادة التصنيع

المحتويات[1]

مواضيع ذات صلة

- نصائح لإصلاح العيوب: القسم 23. 3
- نهج معايرة الشفرة: القسم 25. 6
- التصميم في البناء: الفصل 5
- صفوف ناجحة: الفصل 6
- إجرائيات عالية الجودة: الفصل 7
- البناء التعاوني: الفصل 21
- اختبار المطور: الفصل 22
- مناطق مرجّحة التغيّر: "حدد المناطق المرجحة للتغير" في القسم 5. 3

أسطورة:[2] المشروع البرمجي المُدار بشكل جيد يتدبر شؤون التطوير المنهجي للمتطلبات ويعرّف لائحة مستقرة لمسؤوليات البرنامج. يتبع التصميم المتطلبات، ويُعمل بحذر بحيث يمكن لكتابة الشفرة أن تتابع بشكل خطي، من البداية إلى النهاية، مشيراً إلى أن معظم الشفرة يمكن أن تكتب مرة واحدة وتختبر وتُنسى. بالاستناد إلى

[1] cc2e.com/2436

[2] كل البرمجيات الناجحة تتغير. --فريد بروكس

الأسطورة، الوقت الوحيد الذي يتم فيه تعديل الشفرة بشكل ملحوظ هو خلال طور صيانة البرمجية، وهو شيء ما يحدث فقط بعد تسليم الإصدار الأولي من النظام.

الواقع: تنمو الشفرة بشكل فعلي خلال تطويرها الأولي. العديد من التغيرات التي تُرى خلال كتابة الشفرة الأولية هي على الأقل بدراميّة لتغيرات التي تُرى خلال الصيانة. تستهلك كتابة الشفرة والتصحيح واختبار الوحدة يستهلكن بين 30 و65 بالمئة من الجهد في مشروع اعتيادي، بالاعتماد على حجم المشروع. (انظر الفصل 27، "كيف يؤثر حجم المشروع على البناء،" لتفاصيل.) إذا كانت كتابة الشفرة واختبار الوحدة عمليتين مستقيمتين، لن تستهلكا أكثر من 20-30 بالمئة من الجهد الكلي للمشروع. حتى في المشاريع المُدارة جيداً، على كلٍّ، تتغير المتطلبات بحوالي واحد لأربعة بالمئة في الشهر (جونز 2000). تتغير المتطلبات بثبات مسببة تغيّرات موافقة في الشفرة-أحياناً تغيرات جوهرية في الشفرة.



واقع آخر: تزيد تطبيقات التطوير الحديثة من التركيز على تغيرات الشفرة خلال البناء. في دورات الحياة الأقدم، كان التركيز-أكان ناجحاً أم لا-على تجنب تغيرات الشفرة. تبتعد النُهج الأكثر حداثة عن إمكانية توقع كتابة الشفرة. النُهج الحالية متمحورة على الشفرة، وبتقدم حياة المشروع، تستطيع أن تتوقع نمو الشفرة أكثر من أي توقع.

24. 1 أنواع تطور البرمجيات

يشبه تطور البرمجية التطور في علم الأحياء الذي فيه تكون بعض التحولات مفيدة والعديد من التحولات ليست كذلك. يقدم تطور البرمجية الجيد شفرة يحاكي تطورها الارتقاء من القرود إلى الإنسان البدائي إلى حالتنا السامية الحالية كمطوري برمجيات. أحياناً تسير القوى التطورية ببرنامج على الطريق الآخر، على كلٍّ، محوِّلة البرنامج إلى لولب (حلزون) انحطاط.



التمييز المفتاحي بين نوعي تطور البرمجية هو إن كانت جودة البرنامج تتحسن أو تتراجع تحت التعديلات. إذا أصلحت الأخطاء بشريط لاصق من المنطق وخرافة، ستتراجع الجودة. إذا تعاملت مع التعديلات كفرص لشدّ التصميم الأصلي للبرنامج، ستتحسن الجودة. إن رأيت جودة البرنامج تتراجع، فإن هذا مثل ذلك الكناري الصامت في فتحة المنجم الذي ذكرته سابقاً. إنه تحذير من أن البرنامج يتطور بالاتجاه الخطأ.

تمييز ثان بين نوعي تطور البرمجية هو التمييز بين التغيرات المعمولة خلال البناء والأخرى المعمولة خلال الصيانة. هذان النوعان من التطور يختلفان بعدة طرق. عادة ما تُعمل تغيرات البناء من قبل المطورين الأصليين، وعادة قبل أن ينسى البرنامج بشكل كامل. النظام حتى الآن ليس قيد العمل، لذا الضغط لإنهاء التغيرات هو فقط ضغط جدول المواعيد-ليس 500 مستخدم غاضب يتساءلون لمَ نظامهم عاطل. لنفس السبب، يمكن أن تكون التغيرات خلال البناء سواقة أكثر حرية -النظام في حالة أكثر ديناميكية، وضريبة صنع التغيرات منخفضة. هذه الظروف تقتضي أسلوب لتطور البرمجية يختلف عما ستجد خلال صيانة البرمجية.

فلسفة تطور البرمجية

ضعف شائع في نُهج المبرمجين في طور البرمجية هو أنه يسير كعملية غير مبالية بما حولها.[1] إذا لاحظت أن التطور خلال التطوير أمر محتوم وظاهرة مهمة وخططت له، تستطيع أن تستخدمه لمصلحتك.

التطور هو مجازفة وفرصة للوصول إلى الكمال في نفس الوقت. عندما يتوجب عليك أن تصنع تغيراً، كافح لتحسن الشفرة بحيث تكون التغييرات المستقبلية أسهل. عندما تبدأ بكتابة برنامج لا تكون تعرف أبداً معلومات بقدر التي ستعرفها بعد الانتهاء. عندما تمتلك الفرصة لتنقح برنامجاً، استخدم ما تعلمته لتحسنه. ضع كلاً من شفرتك الأولية وتغييراتك في بالك مضافاً إليها تغيرات أكثر.



القاعدة الرئيسة لتطور البرمجيات هي أن التطور ينبغي أن يحسن الجودة الداخلية للبرنامج. الأقسام التالية تصف كيف تنجز هذا.

2 .24 مقدمة إلى إعادة التصنيع

الاستراتيجية المفتاحية لإنجاز القاعدة الرئيسة في تطور البرمجية هو إعادة التصنيع، والذي عرفه مارتن فولير ب "تغيير يُصنع في البنيان الداخلي للبرمجية ليجعله أسهل للفهم وأقل كلفة للتعديل بدون تغيير سلوكه الظاهر" (فولير 1999). نمت الكلمة "إعادة التصنيع" في البرمجة الحديثة بعيداً عن استخدام لاري قسطنطين الأصلي للكلمة "تصنيع" وفي البرمجة الهيكلية، والتي أشارت إلى تفكيك البرنامج إلى عناصره الأساسية قدر الإمكان (يوردون وقسطنطين 1979).

أسباب إعادة التصنيع

[1] لا توجد شفرة كبيرة أو ملفوفة أو معقدة جداً بحيث لا تستطيع الصيانة أن تجعله أسوأ. --**جيرالد وينبيرغ**

بعض الأحيان تنحطّ الشفرة تحت الصيانة، وبعض الأحيان لا تكون الشفرة جيدة جداً تماماً في المقام الأول. في كلتا الحالتين، توجد نفس علامتي التحذير-أحياناً تدعى "روائح" (فولير 1999) -التي تحدد أين تلزم إعادة التصنيع:

تكون الشفرة مكررة تمثل الشفرة المكررة تقريباً دائماً خلل في تكوين النظام في المقام الأول. تهيئك الشفرة المكررة لتقوم بتعديلات متوازية-متى تقوم بتغييرات في مكان، عليك أن تقوم بتغييرات متوازية في مكان آخر. إنها أيضاً تخرق ما أشار إليه اندريو هنت وديف توماس ب "مبدأ DRY": **لا تكرر نفسك** (2000). أعتقد أن ديفيد بارناس قالها بالشكل الأفضل: "النسخ واللصق هما خطأ تصميمي" (مك كونيل 1998b).

تكون الإجرائية طويلة جداً في البرمجة غرضية التوجه، الإجرائيات الأطول من شاشة هي نادراً ما تلزم وعادة تمثل محاولة لإجبار أن تلبس قدم البرمجة الهيكلية حذاء غرضية التوجه.

تعيّن أحد زبائني لمهمة تجزيء أطول إجرائية في نظام تراثي، والتي كانت أكثر من 12000 سطر طولاً. مع الجهد، استطاع أن يخفّض حجم الإجرائية الأطول إلى حوالي 4000 سطر.

إحدى طرق تحسين نظام هي أن تزيد في وحدويته-تزيد عدد الإجرائيات المعرّفة جيداً والمسماة جيداً والتي تقوم بشيء واحد وتقوم به بطريقة حسنة. عندما تقودك التغييرات إلى إعادة زيارة أقسام من الشفرة، اغتنم الفرصة لتفحص وحدوية الإجرائيات في ذلك القسم. إن كانت الإجرائية ستصبح أنظف إن بُدّل قسم منها إلى إجرائية منفصلة، أنشئ إجرائية مستقلة.

الحلقة طويلة جداً أو معششة جداً تميل أحشاء الحلقة لتكون مرشّحات جيدة للتحويل إلى إجرائيات، والذي يساعد إلى تصنيع أفضل للشفرة ويخفّض تعقيد الحلقة.

يمتلك الصف تماسك ضعيف إذا وجدت صف يأخذ ملكية شرذمة من المسؤوليات غير المترابطة، فينبغي أن يُجزّأ ذلك الصف إلى عدة صفوف، لدى كل منها مسؤوليته عن مجموعة متماسكة من المسؤوليات.

لا تؤمن واجهة الصف مستوى ثابت من التعقيد حتى الصفوف التي تبدأ الحياة بواجهة متماسكة يمكن أن تخسر ثباتها الأصلي. تميل واجهات الصفوف إلى التحول بلطف مع الزمن كنتيجة للتعديلات التي

أُنجزت في نشوة خاطفة والتي تفضّل مصلحة تكامل الواجهة. أخيراً تصبح واجهة الصف وحش صيانة فرانكينستيني[1] والذي يفعل القليل ليُحسّن قابلية الإدارة الفكرية للبرنامج.

لائحة الوسطاء فيها الكثير من الوسطاء تميل البرامج المصنعة بشكل جيد إلى أن تمتلك العديد من إجرائيات صغيرة ومعرفة بشكل جيد والتي لا تحتاج لوائح وسطاء ضخمة. لائحة الوسطاء الطويلة هي تحذير بأن تجريد واجهة الإجرائية لم يُدرس جيداً.

تميل التعديلات في الصف لتكون مقسّمة في إجزاء مستقلة أحياناً يمتلك الصف مسؤوليتين متباينتين أو أكثر. عندما يحدث هذا تجد نفسك تغير إما في قسم في الصف أو في آخر-ما عدا التغييرات القليلة التي تؤثر في كلا الجزأين من الصف. هذه إشارة إلى وجوب تقسيم الصف إلى صفوف متعددة حسب الخطوط المحددة من قبل المسؤوليات المنفصلة.

تتطلب التغييرات تعديلات متوازية في صفوف متعددة رأيت مشروعاً فيه لائحة اختبار بحوالي 15 صف يجب أن تُعدّل متى أُضيف نوع مدخلات جديد. عندما تجد نفسك تقوم بشكل روتيني بتغييرات لنفس المجموعة من الصفوف، فإن هذا يقتضي أن يُعاد ترتيب الشفرة في هذه الصفوف بحيث تؤثر تلك التغييرات بصف واحد فقط. من تجربتي، هذه مثالية صعبة التحقيق، لكنها على أي حال هدف جيد.

يجب أن تعدل هرميات الوراثة بشكل متواز أنْ تجد نفسك تصنع صف ابن من أحد الصفوف في كل مرة تصنع فيها صف فرعي من صف آخر هو نوع خاص من التعديل المتوازي وينبغي أن يعالج.

عبارات* case *يجب أن تعدل بشكل متوازي على الرغم من أن عبارات *case* ليست سيئة بشكل موروث، فإذا وجدت نفسك تصنع تعديلات متوازية في عبارات *case* متشابهة في عدة أقسام من البرنامج، ينبغي أن تسأل نفسك إن كانت الوراثة نهج أفضل.

عناصر البيانات المترابطة والتي تُستخدم معاً ليست منظمة في صفوف إذا وجدت نفسك بشكل متكرر تعالج نفس المجموعة من عناصر البيانات، ينبغي أن تسأل إن كان ينبغي لهذه المعالجات أن تُجمّع في صف لوحدها.

[1] **هامش** فرانكينستين شخصية روائية قامت بخلق وحش قضى عليها في النهاية

تستخدم الإجرائية وظائف أكثر من صف آخر بالمقارنة مع صفها هذا يقترح أنه ينبغي أن تنقل الإجرائية إلى الصف الآخر ومن ثم تستدعيها في صفها القديم.

تم التحميل الزائد لأنماط البيانات البدائية يمكن أن تُستخدم أنماط البيانات البدائية لتمثل عدد غير منته من كيانات العالم الحقيقي. إذا كان برنامجك يستخدم نمط بيانات بدائي مثل العدد الصحيح ليمثل كيان شائع مثل المال، ضع في بالك أن تنشئ صف *Money* بسيط بحيث يستطيع المترجم أن ينجز فحص النمط على متحولات ال *Money*، حتى تستطيع أن تضيف فحوص أمان على القيم المسندة إلى المال، وما إلى هنالك. إذا كان كلا *Money* و*Temperature* أعداد صحيحة، فلن يحذرك المترجم من الإسنادات الخاطئة مثل *bankBalance = recordLowTemperature*.

لا يفعل الصف الشيء الكثير أحياناً تكون نتيجة إعادة تصنيع شفرة ألا يمتلك صف قديم الكثير ليقوم به. إذا لم يكن يبدو أن الصف يحمله وزنه، تساءل إن كان ينبغي أن تعيّن كل مسؤوليات هذا الصف إلى صفوف أخرى وتزيله نهائياً.

سلسلة إجرائيات تمرر بيانات مشردة عندما تجد نفسك تمرر بيانات لإحدى الإجرائيات فقط كي تستطيع أن تمررها إلى إجرائية أخرى تُسمَّى هذه البيانات "بيانات مشردة" (بيج-جونز 1988). قد يكون هذا مقبول، لكن اسأل نفسك إن كان تمرير البيانات المحددة في السؤال متناغم مع التجريد المقدَّم بواسطة كل من واجهتي الإجرائيتين. إن كان التجريد لكلا الإجرائيتين مقبول، فتمرير البيانات مقبول. وإلا، جد مخرجا لتجعل واجهة كل إجرائية أكثر تماسكاً.

لا يقوم كائن وسيط بأي عمل إذا وجدت أن معظم الشفرة في صف تقوم فقط باستدعاءات لإجرائيات في صفوف أخرى، فكر أن كان ينبغي أن تزيل هذه الوساطة وتستدعي الصفوف الأخرى مباشرة.

أحد الصفوف على علاقة حميمة مع آخر بشكل زائد عن الحد ربما يكون التغليف (إخفاء المعلومات) هو أقوى أداة لديك لتجعل البرنامج مُدار فكرياً ولتقلل آثار الارتدادات غير المتوقعة لتغيير الشفرة. في أي وقت ترى صفاً يعرف أكثر مما ينبغي عن آخر-وضمناً، الصفوف الأبناء التي تعرف الكثير عن آبائها-فإن من الأفضل بكثير أن تختار التغليف القوي على الضعيف.

للإجرائية اسم فقير إذا كان للإجرائية اسم فقير، غيّر اسمها في المكان الذي عُرِّفت فيه، غيّر الاسم في كل الأماكن التي استدعيت بها، ومن ثم أعد ترجمة الشفرة. بقدر صعوبة القيام بهذا الآن وأكثر، ستكون صعوبة القيام به لاحقاً، لذا قم به حالما تلاحظ أنه مشكلة.

تكون البيانات الأعضاء عامة البيانات الأعضاء العامة، برأيي، هي دائماً فكرة سيئة. إنها تغبش الخط بين الواجهة والتحقيق، وهي فطرياً تخترق التغليف وتحدّ من المرونة المستقبلية. فكر بجد بإخفاء البيانات الأعضاء العامة خلف إجرائيات الدخول.

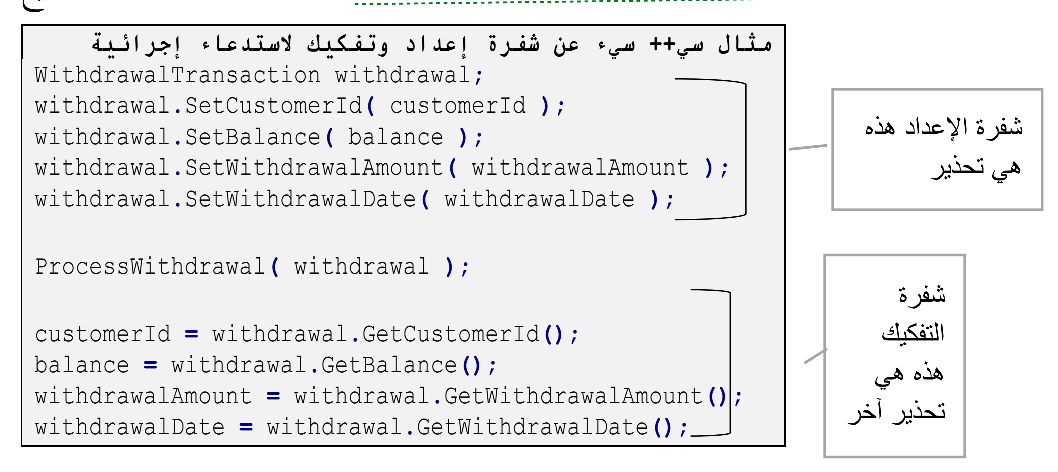
يستخدم صف ابن فقط نسبة صغيرة من إجرائيات آبائه عادةً ما يشير هذا إلى أن الصف الابن أُنشئ لأنها صدفت وحوى الصف الأب الإجرائيات التي يحتاجها الابن، وليس لأن الصف الابن مُتحدِّر بشكل عقلاني عن الصف الأب. فكر بإنجاز تغليف أفضل بتحويل علاقة الصف الابن بأبيه من علاقة "يكون" إلى "يملك"؛ وبدل الصف الأب إلى عضو في الصف الابن سالف الذكر، واكشف فقط الإجرائيات اللازمة بحق في الصف الابن سالف الذكر.

تُستخدم التعليقات لتشرح الشفرة الصعبة للتعليقات دور مهم لتلعبه، لكن ينبغي ألا تستخدم كدعامة لتشرح الشفرة السيئة. الحكمة القديمة الدهور دقيقة: "لا توثّق الشفرة السيئة-أعد كتابتها" (كيرنيغان و بلاوغير 1978).

يتم استخدام البيانات الشاملة[1] عندما تزور مجدداً قسماً من الشفرة يستخدم متحولات شاملة، خذ وقتك وأعد فحصهم. قد تفكر بطريقة لتتجنب استخدام المتحولات الشاملة منذ آخر مرة قمت بزيارة ذلك القسم من الشفرة. لأنك أقل إلفة للشفرة بالمقارنة مع أول كتابة لها، قد تجد الآن أن المتحولات الشاملة تربك بشكل كاف رغبتك بتطوير نهج أنظف. وقد يكون أيضاً أصبح لديك فهم أفضل عن كيفية عزل المتحولات الشاملة في إجرائيات وصول وفهم دقيق للألم الناجم عن ألا تفعل ذلك. عض الرصاصة "تحمل الحالة الصعبة بشجاعة وثبات" واصنع تعديلات مفيدة. ستكون الكتابة الأولية للشفرة بعيدة كفاية في الماضي بحيث تستطيع أن تكون موضوعي بالنسبة لعملك وستكون قريبة كفاية بحيث ستتذكر معظم ما تحتاج لتقوم بالتنقيحات بشكل صحيح. الوقت خلال التنقيحات الأولى هو الوقت المثالي لتحسّن الشفرة.

تستخدم إجرائية ما شفرة إعداد قبل استدعاء إجرائية أو تستخدم شفرة تفكيك بعد استدعاء إجرائية شفرة مثل هذه ينبغي أن تكون تحذيراً لك:

[1] **إشارة مرجعية** من أجل توجيهات في استخدام المتحولات الشاملة، انظر القسم 13. 3، "البيانات الشاملة." من أجل توضيح الفروقات بين البيانات الشاملة وبيانات الصف، انظر "بيانات صف اعتبرت خطأ بيانات شاملة" في القسم 5. 3.



مثال سي++ سيء عن شفرة إعداد وتفكيك لاستدعاء إجرائية

```
WithdrawalTransaction withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );

ProcessWithdrawal( withdrawal );

customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```


تكون إشارة تحذير مشابهة عندما تجد نفسك تنشئ بواني خاصة لصف "مناقلة سحب" والذي يأخذ مجموعة فرعية من بيانات التهيئة العادية الخاصة به بحيث تستطيع أن تكتب شفرة كهذه:

مثل سي++ سيء عن شفرة إعداد وتفكيك لاستدعاء طريقة

```
withdrawal = new WithdrawalTransaction( customerId,
    balance, withdrawalAmount, withdrawalDate );
withdrawal.ProcessWithdrawal();
delete withdrawal;
```

في أي وقت ترى فيه شفرة تهيّئ لاستدعاء إجرائية أو تفككها بعد الاستدعاء، تساءل إن كانت واجهة الإجرائية تمثل التجريد الصحيح. في هذه الحالة، قد ينبغي أن تُعدَّل لائحة وسطاء *ProcessWithdrawal* لكي تدعم الشفرة كالتالي:

مثل سي++ جيد عن إجرائية لا تتطلب شفرة إعداد أو تفكيك

```
ProcessWithdrawal( customerId, balance,
    withdrawalAmount, withdrawalDate );
```

لاحظ أن معكوس هذا المثال يقدم مشكلة مشابهة. إن وجدت نفسك بشكل متكرر لديك كائن WithdrawalTranscation بيدك لكنك محتاج لأن تمرر عدة من قيمه إلى إجرائية مثلما يظهر في هذا المثال، ينبغي أن تفكر بإعادة تصنيع واجهة ProcessWithdrawal بحيث تتطلب كائن WithdrawalTranscation بدلاً من حقوله المستقلة.

مثال سي++ لشفرة تتطلب عدة استدعاءات طرق

```
ProcessWithdrawal( withdrawal.GetCustomerId(), with-
    drawal.GetBalance(), withdrawal.GetWithdraw-
    alAmount(), withdrawal.GetWithdrawalDate() );
```

ولا واحد من هذه النهج يمكن أن يكون صحيحاً، ولا واحد يمكن أن يكون خطأ-إنه يعتمد على إن كان تجريد واجهة *ProcessWithdrawal()* يتوقع أن يكون لديه أربع قطع متباينة من البيانات أو يتوقع أن يكون لديه كائن *WithdrawalTransaction* واحد.

يتضمن البرنامج شفرة تبدو أنها قد تلزم في بعض الأيام المبرمجون سيئون بشكل مشهور في تخمين أي الوظائف قد تلزم يوماً ما. "صمم رأساً" موضوع للعديد من المشاكل القابلة للتوقع:

- متطلبات شفرة "صمم رأساً" لما تُطوَّر بشكل كامل، والذي يعني أن المبرمجين على الأرجح سيخمنون خطأً عن هذه المتطلبات المستقبلية. سيُرمى عمل "صمم رأساً" بعيداً بشكل نهائي.
- إن كان تخمين المبرمج عن المتطلبات المستقبلية قريباً جداً، فلا يزال المبرمج بالعموم بعيد عن توقع كل التعقيدات الخاصة بالمتطلبات المستقبلية. هذه التعقيدات تقوّض افتراضات المبرمج الأساسية للتصميم، والذي يعني ان عمل "صمم رأساً" سيتوجب عليه أن يُرمى بعيداً.
- مبرمجو المستقبل الذين يستخدمون شفرة "صمم رأساً" لا يعرفون أنها شفرة "صمم رأساً،" أو يفترضون أن الشفرة تعمل أفضل مما هي عليه. إنهم يفترضون أن الشفرة قد شُفرت واختبرت وروجعت بنفس مستوى الشفرة الأخرى. إنهم يضيعون الكثير من الوقت في بناء شفرة تستخدم شفرة "صمم رأساً"، فقط ليكتشفوا في النهاية أن شفرة "صمم رأساً" لا تعمل فعلياً.
- تخلق شفرة "صمم رأساً" الإضافية تعقيداً إضافياً، والذي يستدعي اختباراً إضافياً وتصحيح عيوب إضافياً وما إلى ذلك. ويكون الأثر الكلي إبطاء المشروع.

يتفق الخبراء على أن أفضل طريقة لتتجهّز من أجل متطلبات المستقبل هي ليست أن تكتب شفرة تأمليّة، بل هي أن تجعل الشفرة *المطلوبة حالياً* واضحة ومستقيمة قدر الإمكان بحيث سيعرف مبرمجو المستقبل ما تقوم وما لا تقوم بفعله وسيصنعون تغييراتهم تبعاً لذلك (فولير 1999، بيك 2000).

لائحة اختبار: أسباب أن تعيد التصنيع1

- الشفرة مكررة
- إجرائية طويلة جداً
- حلقة طويلة جداً أو معششة بعمق شديد
- لدى صف تماسك ضعيف
- واجهة صف لا تؤمن مستوى متماسك من التجريد
- لائحة وسطاء لديها الكثير جداً من الوسطاء
- التغييرات في الصف تميل لتكون منفصلة ضمن أجزاء مستقلة
- التغييرات تتطلب تعديلات موازية في عدة صفوف
- الهرميات الوراثية يجب أن تُعدل بشكل متواز
- عبارات case يجب أن تُعدل بشكل متواز

[1] cc2e.com/2443

- عناصر البيانات المترابطة التي تُستخدم معاً ليست منظمة في صفوف
- إجرائية تستخدم وظائف من صف آخر أكثر من صفها
- تم التحميل الزائد للأنماط البدائية
- صف لا يقوم بالكثير من العمل
- سلسلة إجرائيات تمرر بيانات مشردة
- كائن وسيط لا يقوم بأي شيء
- أحد الصفوف على علاقة حميمة فوق الحد مع آخر
- لإجرائية اسم فقير
- عناصر البيانات تُعرف كعامة
- صف ابن يستخدم نسبة صغيرة من إجرائيات آبائه
- التعليقات تُستخدم لتشرح الشفرة الصعبة
- يتم استخدام المتحولات الشاملة
- إجرائية تستخدم شفرة إعداد قبل استدعاء إجرائية أو شفرة تفكيك بعد استدعاء إجرائية
- يحتوي البرنامج شفرة تبدو وكأنها ستلزم يوماً ما

أسباب ألا تعيد التصنيع

في اللغة المحكية، تُستخدم "إعادة التصنيع" بفضاضة لتشير إلى إصلاح العيوب، وإضافة وظائف، وتعديل التصميم-بشكل جوهري كمرادف لصنع أي تغيير في الشفرة أياً كان. هذا التخفيف الشائع لمعنى المصطلح بائس. التغيير بذاته ليس فضيلة، لكن التغيير ذو الغاية والمطبَّق مع تمام ملعقة شاي صغيرة من الانضباط، يمكن أن يكون استراتيجية مفتاحية تدعم التحسين الثابت في جودة البرنامج تحت الصيانة وتمنع لولب الموت بمستوى عشوائية البرمجية المألوف جداً كله.

24.3 إعادات التصنيع المخصصة

في هذا القسم، أقدم لائحة عناصر خاصة بإعادات التصنيع، أصف العديد منها بتلخيص الوصف الأكثر تفصيلاً المقدَّم في *Refactoring* (فولير 1999). لم أحاول، على كل، أن أجعل هذه اللائحة مفصلة. بمعنى، كل حالة في هذا الكتاب تبيّن مثال "شفرة سيئة" ومثال "شفرة جيدة" هي مرشحة لتكون إعادة تصنيع. الذي يهمني أني، ركزّتُ على إعادات التصنيع التي وجدتها شخصياً أكثر فائدة.

إعادات التصنيع في مستوى البيانات

إليك إعادات التصنيع التي تحسّن استخدام المتحولات والأنواع الأخرى من البيانات.

بدّل الثوابت المسماة مكان الأرقام السحرية إذا كنت تستخدم أعداد أو سلاسل نصية محرفية مثل *3.14*، استبدل ثابتاً مسمى بتلك المحرفية مثل *PI*.

أعد تسمية متحول باسم أوضح أو أكثر إعلاماً إن لم يكن اسم المتحول واضحاً، غيّره إلى اسم أفضل. تُطبق نفس النصيحة على إعادة تسمية الثوابت والصفوف والإجرائيات، بالطبع.

انقل التعبير إلى السطر استبدل التعبير نفسه بمتحول وسيطي أُسند له التعبير.

استبدل إجرائية بتعبير استبدل إجرائية بتعبير (عادةً بحيث لا يتكرر التعبير في الشفرة).

قدّم متحولاً بينياً اسند تعبيراً إلى متحول بيني يلخص اسمه الغرض من التعبير.

حوّل متحولاً متعدد الاستخدام إلى عدة متحولات وحيدة الاستخدام إذا استخدِم المتحول لأكثر من غرض-متهمون شائعون هم *i, j, temp, x*-أنشئ متحولات منفصلة من أجل كل استخدام، كل منها له اسم أكثر تحديداً.

استخدم متحولاً محلياً للأغراض المحلية بدلاً من وسيط إن كان وسيط دخل فقط يُستخدم كمتحول محلي، أنشئ متحولاً محلياً واستخدمه بدلاً.

حوّل بيانات بدائية إلى صف إن احتاجت بيانات بدائية إلى سلوك إضافي (متضمناً فحص نمط أشدّ) أو بيانات إضافية، حّول البيانات إلى كائن وأضف السلوك الذي تريد. يمكن أن يُطبق هذا على الأنماط الرقمية البسيطة مثل Money و Temperature المال ودرجة الحرارة. ويُطبق على الأنماط التعدادية مثل Color, Shape, OutputType اللون والشكل ونمط الخرج.

حوّل مجموعة شفرات نمط إلى صف أو تعدادية في البرامج الأقدم، من الشائع أن ترى ارتباطات مثل

```
const int SCREEN = 0;
const int PRINTER = 1;
const int FILE = 2;
```

بدلاً من تعريف ثوابت منفردة، أنشئ صفاً بحيث تستطيع أن تتلقى منافع فحص النمط الأشد وتهيئ نفسك لتقدم دلالات أغنى ل*OutoutType* إذا بالعمر احتجت لذلك. إنشاء تعدادية في بعض الأحيان بدل جيد لإنشاء صف.

حوّل مجموعة شفرات نمط إلى صف مع صفوف فرعية إن كانت للعناصر المختلفة المرتبطة بأنواع مختلفة سلوكيات مختلفة، فكر بإنشاء صف أب للنمط مع صفوف أبناء لكل شفرة نمط. من أجل الصف الأب *OutputType*، قد تنشئ صفوف أبناء مثل *Screen, Printer, File*.

استبدل كائن بمصفوفة إن كنت تستخدم مصفوفة فيها عناصر مختلفة ترجع إلى أنماط مختلفة، أنشئ كائن يملك حقل لكل عنصر مشكّل للمصفوفة.

غلّف مجموعة إن كان الصف يقدم مجموعة-هذه تعني، مصفوفة أو مكدس أو رتل أو ما شابه-ولديه عدة نسخ من المجموعة تطوف في الأرجاء، فإن هذا يمكن أن يخلق صعوبات في التزامن. فكر بأن يكون ما يقدمه الصف مجموعة بخاصية للقراءة فقط، وزود إجرائيات لتضيف أو تزيل عناصر من المجموعة.

استبدل صف بيانات بسجل تراثي أنشئ صفاً يحتوي عناصر السجل. يسمح لك إنشاء صف أن تجعل فحص الخطأ مركزياً، والثبات، والعديد من العمليات التي تقلق السجل

إعادات التصنيع في مستوى العبارة

هنا إعادات التصنيع التي تحسّن استخدام العبارات المنفصلة.

فكك تعبيراً منطقياً بسّط تعبيراً منطقياً بتقديم متحولات وسيطة مسماة بشكل جيد والتي تساعد في توثيق معنى التعبير.

انقل تعبير منطقي معقد إلى تابع منطقي مسمى بشكل جيد إن كان التعبير معقداً كفاية، إعادة التصنيع هذه تستطيع أن تحسن قابلية القراءة. إن استُخدِم التعبير أكثر من مرة، فإن هذا يتخلص من الحاجة للتعديلات المتوازية ويخفف من فرصة خطأ في استخدام التعبير.

وحّد الشظايا المكررة في الأجزاء المختلفة من شرطيّة إن كان لديك نفس الأسطر من الشفرة مكررة في نهاية كتلة *else*، والتي هي موجودة في نهاية كتلة if، انقل هذه الأسطر بحيث تأتي بعد كتلة if-then-else كلها.

استخدم break أو return بدلاً من متحول التحكم بالحلقة إن كان لديك متحول ضمن الحلقة مثل *done* والذي يُستخدم للتحكم بالحلقة، استخدم *break* أو *return* لتخرج من الحلقة بدلاً.

عُد فور معرفتك الجواب بدلاً من إسناد قيمة إرجاع ضمن عبارات if-then-else معششة تكون الشفرة غالباً أسهل للقراءة وأقل ميلاً إلى الأخطاء إذا خرجت من الإجرائية فور معرفتك لقيمة الإرجاع. البديل بوضع قيمة إرجاع ومن ثم فكفكة طريقك عبر الكثير من المنطق يمكن أن يكون أصعب تَتبُّعاً.

استبدل تعدد الأشكال بالشرطيات (خصوصاً عبارات case المكررة) الكثير من المنطق الذي جرت العادة بأن يضمن في عبارات *case* في البرامج المهيكلة يمكن بدلاً أن يخبز ليصبح هرمية وراثية ويُتمم عبر استدعاءات متعددة الأشكال لإجرائية.

أنشئ واستخدم كائنات فارغة بدلاً من اختبار القيم الفارغة null أحياناً يكون لدى كائن فارغ سلوك عام أو بيانات مرتبطة به، مثل الإشارة إلى مواطن اسمه غير معروف ب "مواطن." في هذه الحالة، فكر بنقل مسؤولية التعامل مع القيم الفارغة (null) خارج الشفرة الزبونة وإلى الصف-هذا يعني، تجعل الصف *Customer* يعرّف المواطن غير المعروف ب "مواطن" بدلاً من جعل الشفرة الزبونة ل*Customer* تختبر بشكل متكرر إن كان اسم الزبون غير معروف وتستبدل "مواطن" إن لم يكن معروفا.

إعادات التصنيع في مستوى الإجرائية

إليك هنا إعادات التصنيع التي تحسّن الشفرة في مستوى الإجرائية المستقلة.

استخلص إجرائية/استخلص طريقة أزل الشفرة "في الخط" من إجرائية، وحولها إلى إجرائيتها الخاصة.

انقل شفرة إجرائية "إلى الخط" خذ شفرة من إجرائية جسمها بسيط ويشرح نفسه بنفسه، وانقل شفرة الإجرائية تلك "إلى الخط" إلى مكان استخدامها.

حوّل إجرائية طويلة إلى صف إن كانت الإجرائية طويلة جداً، فأحياناً تحويلها إلى صف ومن ثم التفكيك الإضافي للإجرائية المشكّلة لتصبح عدة إجرائيات سيحسّن قابلية القراءة.

استبدل خوارزمية بسيطة بخوارزمية مقعدة بدل خوارزمية بسيطة مكان أخرى معقدة.

أضف وسيطاً إن احتاجت الإجرائية معلومات أكثر عن مستدعيها، أضف وسيط بحيث يمكن أن تُزوِّد تلك المعلومات.

أزل وسيطاً إن لم تعد إجرائية تستخدم وسيطاً، أزله

افصل عمليات الاستعلام عن عمليات التعديل بشكل طبيعي، لا تغير عمليات الاستعلام حالة كائن. إذا غيّرت عملية مثل *GetTotals()* من حالة الكائن، افصل وظيفة الاستعلام عن وظيفة تغيير الحالة وأمّن إجرائيتين منفصلتين.

اجمع إجرائيتين متشابهتين عن طريق تمرير الوسطاء إليهما قد تختلف إجرائيتين متشابهتين فقط فيما يتعلق بقيمة ثابتة تُستخدم ضمن الإجرائية. اجمع الإجرائيتين إلى إجرائية واحدة، ومرر القيمة المستخدَمة كوسيط.

افصل الإجرائيات التي يعتمد سلوكها على وسطاء تُمرَّر لها إن كانت إجرائية تنفذ شفرات متباينة حسب قيمة وسيط دخل، فكر بتكسير الإجرائية إلى إجرائيات منفصلة يمكن أن تُستدعى بشكل منفصل، دون تمرير وسيط الدخل المعين ذاك.

مرر كامل الكائن بدلاً من حقول محددة إن وجدت نفسك تمرر عدة قيم من نفس الكائن إلى إجرائية، فكر بتغيير واجهة الإجرائية بحيث تأخذ كامل الكائن بدلاً.

مرر حقولاً محددة بدلاً من كامل الكائن إن وجدت نفسك تنشئ كائناً فقط حتى تستطيع أن تمرره إلى إجرائية، فكر بتعديل الإجرائية بحيث تأخذ حقولاً محددة فقط بدلاً من كامل الكائن.

غلّف "التحويل الصريح للأنماط نزولاً" إذا أرجعت إجرائية كائناً، ينبغي بشكل طبيعي أن تعيد النمط الأكثر تحديداً خ للكائن حسبما تعلم عن ذلك. هذا قابل للتطبيق بشكل خاص على الإجرائيات التي ترجع مكررات (iterators) أو مجموعات (collections)أو عناصر مجموعات وما إلى هنالك.

إعادات التصنيع المتعلقة بتحقيق الصف

إليك إعادات التصنيع التي تحسّن في مستوى الصف

حوّل كائنات القيمة إلى كائنات المرجع إن وجدت نفسك تنشئ وتصون عدة نسخ من كائنات كبيرة أو معقدة، غيّر استخدامك لهذه الكائنات بحيث توجد نسخة سيدة واحدة فقط (كائن القيمة) وبقية الشفرة تستخدم مراجع إلى ذلك الكائن (كائنات المرجع).

حوّل كائنات المرجع إلى كائنات القيمة إن وجدت نفسك تقوم بالكثير من أعمال تدبير المراجع لكائنات صغيرة أو بسيطة، بدّل استخدامك لتلك الكائنات بحيث تكون كل الكائنات كائنات قيمة.

استبدل تهيئة البيانات بإجرائيات افتراضية إن كان لديك مجموعة صفوف أبناء تختلف فقط فيما يتعلق بقيم ثابتة يعيدونها، بدلاً من إنشاء إجرائيات تجاوز-override-أعضاء في الصفوف الأبناء، اجعل الصفوف الأبناء تهيئ الصف بقيم ثابتة مناسبة، ومن ثم أنشئ شفرة عامة في الصف الأب تعمل مع هذه القيم.

غيّر إجرائية عضو أو تموضع بيانات فكر بصنع عدة تغييرات عامة في الهرمية الوراثية. هذه التغييرات تُنجَز بشكل طبيعي لتزيل التكرار في الصفوف الأبناء:

- اسحب إجرائية صعوداً إلى الصف الأب الخاص بها.
- اسحب حقلاً صعوداً إلى الصف الأب الخاص به.
- اسحب جسم بانٍ صعوداً إلى الصف الأب الخاص به.

تُصنع عدة تغييرات أخرى لتدعم التخصيص في الصفوف الأبناء:

- ادفع إجرائية نزولاً إلى الصف الابن الخاص بها.
- ادفع حقلاً نزولاً إلى الصف الابن الخاص به.
- ادفع جسم بانٍ نزولاً إلى الصف الابن الخاص به.

استخرج شفرة مخصصة إلى صف ابن إن كان للصف شفرة تُستخدم فقط من قبل مجموعة من نسخه، انقل هذه الشفرة المخصصة إلى الصف الابن خاصتها.

اجمع شفرة متشابهة إلى صف أب إن كان لدى صفيين ابنين شفرة متشابهة، اجمع تلك الشفرة وانقلها إلى الصف الأب.

إعادات التصنيع على مستوى واجهة الصف

إليك إعادات التصنيع التي تذهب في مسار واجهات صفوف أفضل

انقل إجرائية إلى صف آخر أنشئ إجرائية جديدة في الصف الهدف وانقل جسم الإجرائية من الصف المصدر إلى الصف الهدف. من ثم تستطيع أن تستدعي الإجرائية الجديدة من الإجرائية القديمة.

حوّل صفاً إلى صفين إن كان للصف اثنتين أو أكثر من مناطق المسؤولية المتباينة، اكسر الصف إلى عدة صفوف، كل منها له مسؤولية مُعرَّفة بوضوح.

استغنِ عن صف إن لم يكن الصف يقوم بالكثير من العمل، انقل شفرته إلى صفوف أخرى أكثر تماسكاً واستغن عن الصف.

اخف تفويضاً أحياناً يستدعي الصف أ الصف ب والصف ج، عندما ينبغي فعلياً أن يستدعي الصف أ الصف ب فقط والصف ب ينبغي أن يستدعي الصف ج. اسأل نفسك ما هو التجريد الصحيح لتفاعل أ مع ب. إن كان ينبغي أن يكون ب مسؤولاً عن استدعاء ج، دع ب يستدعي ج.

أزل كائناً بينياً إن كان الصف أ يستدعي الصف ب والصف ب يستدعي الصف ج، أحياناً تجري الأمور بشكل أفضل إذا جعلت الصف أ يستدعي الصف ج مباشرة. التساؤل عن إن كان ينبغي عليك أن تفوض إلى الصف ب يعتمد على ما سيحافظ بالشكل الأفضل على وحدة واجهة الصف ب.

استبدل التفويض بالوراثة إن كان صف يحتاج صفاً آخر لكنه يريد تحكم أكثر بواجهته، اجعل الصف الأب حقلاً في الصف الابن سابق الذكر ومن ثم اكشف مجموعة الإجرائيات التي ستؤمن تجريداً متماسكاً.

استبدل الوراثة بالتفويض إن كان صف يكشف كل الإجرائيات العامة لصف مُفوَّض (صف عضو)، رث (الأمر ورث يرث) من الصف المفوض بدلاً من مجرد استخدامه.

قدّم إجرائية أجنبية إن كان الصف يحتاج إجرائية إضافية وأنت لا تستطيع تعديل الصف لتؤمنها، فإنك تستطيع إنشاء إجرائية جديدة ضمن الصف الزبون تؤمن تلك الوظيفة.

قدّم صف امتداد إن كان الصف يحتاج عدة إجرائيات إضافية وأنت لا تستطيع تعديل الصف، فإنك تستطيع إنشاء صف جديد يجمع وظائف الصف غير القابلة للتعديل مع وظائف إضافية. تستطيع أن تفعل هذا إما بإنشاء صف ابن من الصف الأصلي وإضافة الإجرائيات الجديدة أو بتغليف الصف وكشف الإجرائيات التي تحتاج.

غلّف متحولاً عضواً مكشوفاً إن كانت واحدة من البيانات الأعضاء عامةً، غيرها إلى خاصة واكشف قيمتها عبر إجرائية بدلاً.

أزل الإجرائيات Set() للحقول التي لا يمكن أن تُغيّر إن كان من المفترض لحقل أن يُعيّن وقت إنشاء الكائن ولا يُغيَّر بعدها، هيء هذا الحقل في باني الكائن بدلاً من تأمين إجرائية *Set()* مضللة.

خبّئ الإجرائيات غير المصمَّمة للاستخدام خارج الصف إن كان على واجهة الصف أن تكون أكثر تماسك بدون إجرائية، خبئ الإجرائية.

غلّف الإجرائيات غير المستخدمة إن وجدت نفسك تستخدم بشكل متكرر جانباً فقط من واجهة صف، أنشئ واجهة جديدة للصف تكشف تلك الإجرائيات الضرورية فقط. تأكد أن الواجهة الجديدة تؤمن تجريداً متماسكاً

اجمع صف أب وصف ابن إن كان تحقيقهما متشابهاً جداً إن لم يقدم صف ابن اختصاصاً مهماً، ادمجه بالصف الأب الخاص به.

إعادات التصنيع على مستوى النظام

إليك هنا إعادات التصنيع التي تحسن الشفرة على مستوى النظام كاملاً:

أنشئ مصدر مرجعي حاسم للبيانات التي لا تستطيع التحكم بها أحياناً تكون لديك بيانات مُصانة من قبل النظام بحيث لا تستطيع الوصول إليها بشكل منسجم أو ملائم من كائنات أخرى تحتاج أن تعرف عن تلك البيانات. مثال شائع هو البيانات المصانة في عنصر تحكم **واجهة المستخدم الرسومية** GUI. في هكذا حالة، تستطيع إنشاء صف يعكس صورة البيانات في عنصر التحكم ذاك، وبعدها تجعل كلاً من عنصر التحكم والشفرة الأخرى تتعامل مع الصف كمصدر حاسم لتلك البيانات.

غيّر ارتباط صف أحادي الاتجاه إلى ارتباط صف ثنائي الاتجاه إن كان لديك صفين يحتاجان أن يستخدما ميزات بعضهم البعض لكن صف واحد فقط يعرف عن الصف الآخر، غّير الصفين بحيث يعرف كلاهما عن الآخر.

غيّر ارتباط صف ثنائي الاتجاه إلى ارتباط صف أحادي الاتجاه إن كان لديك صفيين يعرف كلاهما عن ميزات الآخر لكن صف واحد فقط يحتاج فعلياً أن يعرف عن الآخر، غير الصفين بحيث يعرف واحد عن الآخر بدون العكس.

أمّن طريقة مصنع بدلاً من بانٍ بسيط استخدم طريقة (إجرائية) مصنع عندما تريد إنشاء كائنات معتمدة على شفرة نمطية أو عندما تريد أن تعمل مع كائنات مرجع بدلاً من كائنات قيم.

استبدل استثناءات بشفرات معالجة الخطأ أو العكس حسب استراتيجية التعامل مع الخطأ الخاصة بك، تأكد أن الشفرة تستخدم نهج معياري.

لائحة اختبار: ملخص لإعادات التصنيع[1]

إعادات التصنيع على مستوى البيانات

- استبدل الثوابت المسماة بالأرقام السحرية.
- أعد تسمية متحول باسم أكثر وضوح أو أكثر إعلام.
- انقل تعبير "إلى السطر."
- استبدل إجرائية بتعبير.
- قدّم متحولاً بينياً.
- حوّل متحول متعدد الاستخدام إلى عدة متحولات وحيدة الاستخدام.
- استخدم متحولاً محلياً لأغراض محلية بدلاً من وسيط.
- حوّل بيانات بدائية إلى صف.
- حوّل مجموعة شفرات نمط إلى صف أو تعدادية.
- حوّل مجموعة شفرات نمط إلى صف مع صفوف فرعية (أبناء).
- غيّر مصفوفة إلى كائن.
- غلّف مجموعة (مكدس، مصفوفة ...)
- استبدل بيانات صف بسجل تقليدي.

إعادات التصنيع على مستوى العبارة

- فكك تعبيراً منطقياً.
- انقل تعبيراً منطقياً معقداً إلى تابع منطقي مسمى بشكل جيد.
- ادمج الشظايا المتكررة في أجزاء مختلفة من شرطية.
- استخدم break أو return بدلاً من متحول التحكم بالحلقة.
- ارجع حالما تعرف الجواب بدلاً من تعيين قيمة إرجاع ضمن عبارات if-then-else معششة.
- استبدل تعدد الأشكال بشرطيات (على وجه الخصوص عبارات case مكررة).
- أنشئ واستخدم كائنات خالية بدلاً من اختبار القيم الخالية (null).

إعادات التصنيع على مستوى الإجرائية

- استخرج إجرائية
- انقل شفرة إجرائية "إلى السطر."
- حوّل إجرائية طويلة إلى صف.

[1] cc2e.com/2450

- بدّل خوارزمية بسيطة مكان خوارزمية معقدة.
- أضف وسيطاً.
- أزل وسيطاً.
- افصل عمليات الاستعلام عن عمليات التعديل.
- ادمج الإجرائيات المتشابهة بضبط وسطائها.
- جزّء الإجرائية التي يعتمد سلوكها على وسطاء تمرر لها.
- مرر كائناً كاملاً بدلاً من حقول محددة.
- مرر حقولاً محددة بدلاً من كائن كامل.
- غلّف "التحويل القسري الصريح إلى صف ابن."

إعادات التصنيع على مستوى الصف

- غيّر كائنات القيمة إلى كائنات المرجع.
- غيّر كائنات المرجع إلى كائنات القيمة.
- استبدل تهيئة البيانات بإجرائيات افتراضية.
- غيّر إجرائية عضو أو تموضع بيانات.
- استخرج شفرة متخصصة إلى صف فرعي.
- ادمج الشفرة المتشابهة إلى الصف الأب.

إعادات التصنيع على مستوى واجهة الصف

- انقل إجرائية إلى صف آخر.
- حوّل صفاً إلى صفين.
- استغن عن صف.
- خبئ التفويض.
- أزل كائناً بينياً.
- استبدل التفويض بالوراثة.
- استبدل الوراثة بالتفويض.
- قدّم إجرائية أعجمية.
- قدّم صف امتداد.
- غلّف متحول عضو مكشوف.
- أزل الإجرائيات ()Set للحقول التي لا يمكن أن تتغير.

- خبئ الإجرائيات غير المعدّة للاستخدام خارج الصف.
- غلّف الإجرائيات غير المستخدمة.
- ادمج صفاً أباً وصفاً ابناً إن كان تحقيقهما متشابهاً جداً.

إعادات التصنيع على مستوى النظام

- أنشئ مصدر مرجعي حاسم للبيانات التي لا تستطيع التحكم بها.
- غيّر ارتباط صف وحيد الاتجاه إلى ارتباط صف ثنائي الاتجاه.
- غيّر ارتباط صف ثنائي الاتجاه إلى ارتباط صف وحيد الاتجاه.
- أمّن إجرائية مصنع بدلاً من بان بسيط.
- استبدل استثناءات بشفرات خطأ أو العكس.

4.24 إعادة التصنيع بأمان

إعادة التصنيع تقنية قوية لتحسين جودة الشفرة[1]. ككل الأدوات القوية، يمكن أن تسبب إعادة التصنيع أذى أكثر من نفع إن استخدمت بشكل خاطئ. بضع توجيهات بسيطة يمكن أن تمنع الخطوات الخاطئة في إعادة التصنيع:

احفظ الشفرة التي تبدأ بها قبل أن تبدأ بإعادة التصنيع، تأكد أنك تستطيع العودة إلى الشفرة التي بدأت بها. احفظ إصداراً في نظام التحكم بالإصدار خاصتك، أو انسخ الملفات الصحيحة إلى مسار الاحتياط.

حافظ على إعادات التصنيع صغيرة بعض إعادات التصنيع أضخم من بعض، وما يشكّل "إحدى إعادات التصنيع" بالضبط يمكن أن يكون غامضاً قليلاً. حافظ على إعادات التصنيع صغيرة بحيث تستطيع أن تفهم بشكل كامل كل تأثيرات التغييرات التي أجريتها. تقدم إعادات التصنيع التفصيلية التي وُصفت في *Refactoring* (فولير 1999) العديد من الأمثلة الجيدة عن كيفية عمل هذا الأمر.

1 فتح نظام قيد العمل أكثر شبهاً بفتح دماغ انسان وتبديل عصب من فتح بالوعة وتبديل قسطل مغسلة. هل ستكون الصيانة أسهل إن سميناها "عملية جراحية لدماغ البرمجية؟" --جيرالد وينبرغ

قم بإعادات التصنيع واحدة في كل مرة بعض إعادات التصنيع أعقد من الأخرى. من أجل كل إعادات التصنيع ما عدا الأبسط، قم بإعادات التصنيع واحدة في كل مرة، وإعادة الترجمة وإعادة الاختبار بعد إعادة تصنيع قبل القيام بالتالية.

اصنع لائحة بالخطوات التي تنوي فعلها امتداد طبيعي **لعملية البرمجة بالشفرة الزائفة** أن تصنع لائحة بإعادات التصنيع التي ستنقلك من **النقطة أ** إلى **النقطة ب**. يساعدك إنشاء لائحة على الحفاظ على كل تغيير في السياق.

اصنع باحة وقوف عندما تكون في منتصف الطريق عبر إحدى إعادات التصنيع، تجد أحياناً أنك تحتاج إعادة تصنيع أخرى. في منتصف الطريق عبر إعادة التصنيع تلك، تجد إعادة تصنيع ثالثة ستكون مفيدة. من أجل التغييرات غير اللازمة حالاً، اصنع "باحة وقوف،" لائحة بالتغييرات التي ترغب القيام بها في نقطة ما لكن لا تحتاج أن تُقام الآن تماماً.

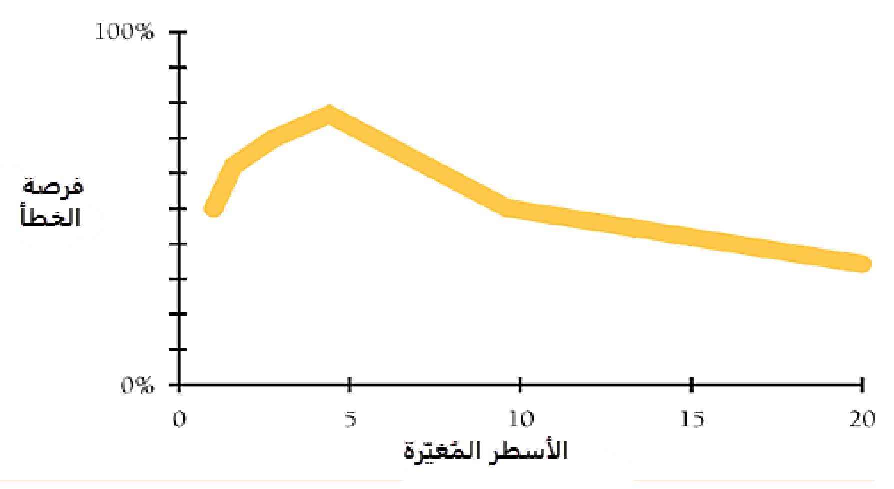
اصنع نقاط علّام متكررة من السهل أن تجد الشفرة فجأة سارت بانحراف بينما تقوم بإعادة التصنيع. بالإضافة إلى حفظ الشفرة التي بدأت بها، احفظ نقاط علّام في خطوات عديدة في جلسة إعادة التصنيع بحيث تستطيع العودة إلى برنامج شغّال إن شفرّت بنفسك إلى نهاية ميتة.

استخدم تحذيرات مترجمك من السهل صنع أخطاء صغيرة تفلت من المترجم. تعيير مترجمك إلى مستوى التحذير الأكثر انتقائية الممكن سيساعد بالقبض على العديد من الأخطاء تقريباً حالما تكتبها.

أعد الاختبار ينبغي أن تُتمّم تغيرات الشفرة بإعادات الاختبار. طبعاً، يرتبط هذا بوجود مجموعة جيدة من حالات الاختبار في المقام الأول. اختبار التراجع ومواضيع اختبار أخرى وُصفت بتفصيل أكثر في الفصل 22، "اختبار المطور."

أضف حالات اختبار بالإضافة إلى إعادة الاختبار بواسطة اختباراتك القديمة، أضف اختبارات وحدة جديدة لتدرّب الشفرة الجديدة. أزل أية حالات اختبار أصبحت مهملة نتيجة لإعادات التصنيع.

راجع التغييرات [1] إن كانت التغييرات مهمة من المرة الأولى وصاعداً، فإنها حتى أكثر أهمية خلال التعديلات اللاحقة. كتب إد يوردون تقريراً عن أن المبرمجين عادةً يملكون فرصة بأكثر من 50 بالمئة لصنع خطأ في محاولتهم الأولى لصنع تغيير (يوردون 1986b). ومما يثير المتعة، إن عمل المبرمجون على جانب جوهري من الشفرة، بدلاً من عدة أسطر فقط، فإن الفرصة لصنع تعديل صحيح تتحسن، كما يظهر في الشكل 1-24. على وجه الخصوص، بتغير الأسطر، عند زيادتها من 1 إلى 5 أسطر تزداد الفرصة لصنع تغيير سيئ. بعد ذلك، الفرصة لصنع تغيير سيئ تتناقص.



الشكل 1-24 تتجه التغييرات الصغيرة لتكون ميالة أكثر إلى الخطأ من التغييرات الكبيرة (وينبرغ 1983)

يتعامل المبرمجون مع التغييرات الصغيرة بشكل عفوي. لا يقومون بالفحص المكتبي لها، ولا يدعون الآخرين لمراجعتها، وأحياناً لا يشغّلون الشفرة ليتأكدوا أن الإصلاح يعمل بشكل صحيح.

العبرة بسيطة: عامل التغييرات البسيطة كما لو كانت معقدة. إحدى المنظمات التي استعملت المراجعات لتغييرات السطر الواحد وجدت أن معدل أخطائها سار من 55 بالمئة قبل المراجعات إلى 2 بالمئة بعدها (فريدمان ووينبرغ 1982). سارت منظمة اتصالات بنسبة صحة من 86 بالمئة قبل مراجعة تغييرات الشفرة إلى 99.6 بالمئة بعدها (بيروت 2004).

اضبط نهجك حسب مستوى خطورة التصنيع بعض إعادات التصنيع أخطر من بعض. إعادة تصنيع مثل "استبدل ثابتاً مسمى برقم سحري" نسبياً عديمة الخطر. إعادات التصنيع التي تتضمن تغييرات لواجهة صف أو إجرائية، أو تغييرات لمخطط قاعدة البيانات، أو تغييرات للاختبارات المنطقية، من بين الكل، تميل لتكون أكثر خطورة. من أجل إعادات تصنيع أسهل، قد تنظم عملية إعادة التصنيع الخاصة بك لتقوم بأكثر من إعادة تصنيع واحدة في المرة الواحدة ولتعيد الاختبار ببساطة، بدون السير خلال مراجعة رسمية.

[1] **إشارة مرجعية** لتفاصيل حول المراجعات، انظر الفصل 21، "البناء التعاوني."

من أجل إعادات تصنيع أخطر، توخى الحذر بدلاً من أن تجازف. قم بإعادات التصنيع واحدة في المرة الواحدة. دع شخصاً غيرك يراجع إعادة التصنيع أو استخدم البرمجة الثنائية لإعادة التصنيع تلك، بالإضافة إلى فحوصات المترجم العادية واختبارات الوحدة.

أوقات سيئة لتقوم بإعادة تصنيع

إعادة التصنيع تقنية قوية، لكنها ليست دواء لكل الأمراض وهي عرضة لبضعة أنواع محددة من إساءة الاستخدام.

لا تستخدم إعادة التصنيع كغلاف وإصلاح للشفرة[1] المشكلة الأسوأ في إعادة التصنيع هي كيف يُساء استخدامها. سيقول المبرمجون أحياناً أنهم يقومون بإعادة تصنيع، عندما يكون كل ما يقومون به هو تصحيح الشفرة بالفعل، آملين أن يجدوا طريقاً لجعلها تعمل. تشير إعادة التصنيع إلى التغييرات الناجحة في الشفرة الشغالة التي تعمل والتي لا تؤثر على سلوك البرنامج. المبرمجون الذين يصححون شفرة عاطلة لا يقومون بإعادة تصنيع؛ إنهم يقرصنون إعادة التصنيع.

تجنب إعادة التصنيع كبدل لإعادة الكتابة[2] أحياناً لا تحتاج الشفرة تغييرات طفيفة-إنها تحتاج أن تُقذف خارجاً بحيث تستطيع البدء من جديد. إن وجدت نفسك في جلسة إعادة تصنيع عظيمة، اسأل انفسك إن كان ينبغي عليك بدلاً أن تعيد التصميم والتحقيق لذلك القسم من الشفرة على أرض خالية.

5.24 استراتيجيات إعادة التصنيع

عدد إعادات التصنيع التي ستكون مفيدة لأي برنامج محدد هو غير محدود حقيقةً. إعادة التصنيع هي موضع لنفس قانون "إرجاعات التقليص"[3] حالها حال بقية النشاطات البرمجية، وقاعدة 80\20 تُطبَّق. اصرف وقتك في القيام ب 20 بالمئة من إعادات التصنيع التي تؤمن 80 بالمئة من الفائدة. اعتبر التوجيهات التالية عندما تقرر أي إعادة تصنيع هي الأكثر أهمية:

---

[1] لا تكتب وظيفة بشكل جزئي بنية أن تعيد التصنيع لتكملها لاحقاً. --جون مونزو

[2] إعادة التصنيع الكبيرة هي وصفة للدمار. --كنت بيك

[3] **مصطلحات** إرجاعات التقليص (diminishing returns) هو مبدأ ينص على: بعد نقطة محددة، زيادة عامل واحد والإبقاء على بقية عوامل الإنتاج ثابتة سيسبب في النهاية عائداً أقل.

أعد التصنيع عندما تضيف إجرائية عندما تضيف إجرائية، افحص أن كانت الإجرائيات ذات الصلة منظمة بشكل جيد. إن لم تكن، أعد تصنيعها.

أعد التصنيع عندما تضيف صفاً غالباً ما تحضر إضافة صف مشاكلَ في الشفرة الموجودة إلى السطح. استخدم هذا التوقيت كمناسبة لتعيد تصنيع الصفوف الأخرى ذات الصلة القريبة بالصف الذي تضيفه.

أعد التصنيع عندما تصلح عيباً استثمر الفهم الذي اكتسبته من إصلاح عيب في تحسين الشفرة الأخرى التي قد تكون ميالة إلى عيوب مشابهة.

استهدف الوحدات الميالة للخطأ[1] بعض الوحدات أكثر هشاشة وميلاناً إلى الخطأ من الأخرى. هل يوجد قسم في الشفرة تخاف منه أنت وكل شخص في فريقك؟ هذا قد يكون وحدة ميالة للخطأ. على الرغم من أن النزعة الطبيعية لمعظم الناس أن تتجنب هذه الأقسام المتحدّية من الشفرة، فإن استهداف هذه الأقسام لإعادة تصنيعها يمكن أن يكون واحدة من الاستراتيجيات الأكثر فعالية (جونز 2000).

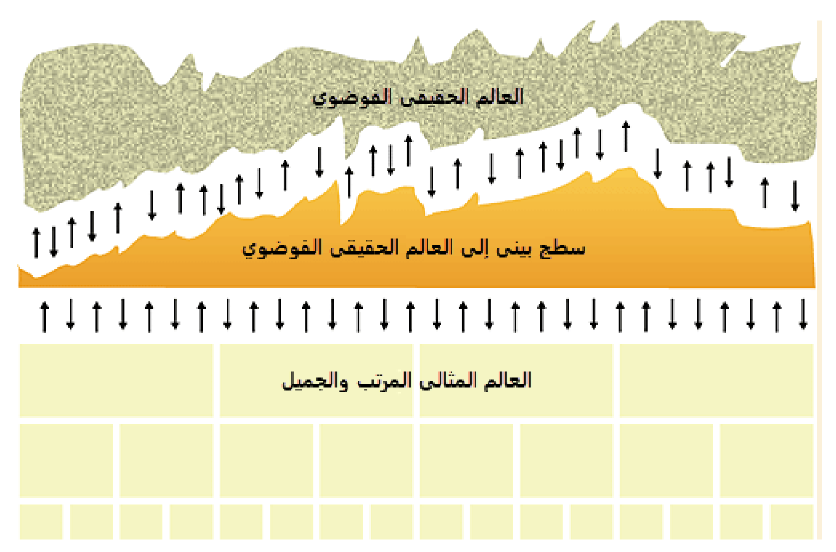
استهدف الوحدات عالية التعقيد نهجٌ آخر أن تركز على الوحدات التي تمتلك معدلات التعقيد الأعلى. (انظر "كيف تقيس التعقيد" في القسم 19. 6 لتفاصيل عن هذه القياسات.) وجدت دراسة كلاسيكية أن جودة البرنامج تحسنت بشكل عظيم عندما ركز مبرمجو الصيانة جهود تحسيناتهم على الوحدات التي امتلكت التعقيد الأعلى (هنري وكافورا 1984).

في بيئات الصيانة، حسّن الأجزاء التي تلامسها الشفرة التي لم تُعدَّل مطلقاً لا تحتاج أن يُعاد تصنيعها. لكن عندما تلامس حقاً قسماً من الشفرة، تأكد من أن تتركه بحال أفضل مما وجدته عليه.

عرف سطحاً بينياً بين الشفرة النظيفة والشفرة القبيحة، ثم انقل الشفرة عبر هذا السطح "العالم الحقيقي" أكثر فوضى مما تظن. قد تأتي الفوضى من قواعد العمل المعقدة، أو واجهات العتاد الصلب، أو الواجهات البرمجية. مشكلة شائعة في الأنظمة الهرمة هي أنّ شفرة الإنتاج المكتوبة بطريقة فقيرة يجب حتماً أن تبقى جاهزة للعمل في كل الأوقات.

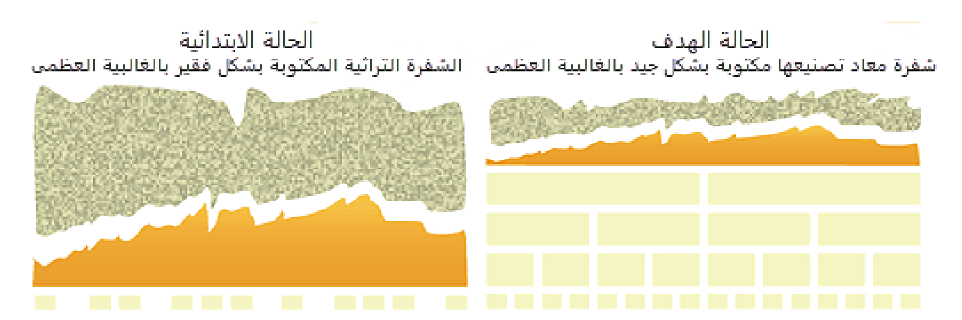
[1] **إشارة مرجعية** لتفاصيل حول الشفرة الميالة إلى الخطأ، انظر "أي الصفوف تحوي أخطاء أكثر؟" في القسم 22. 4

استراتيجية فعالة لاستعادة شباب تلك الأنظمة الهرمة هي أن تعيّن بعض الشفرة ككائنات في العالم الحقيقي الفوضوي، وبعض الشفرة ككائنات في العالم الجديد المثالي، وبعض الشفرة ككائنات في السطح البيني بين الاثنين. يوضح الشكل 24-2 هذه الفكرة.



الشكل 24-2 لا يتوجب على شفرتك أن تكون فوضوية لأن العالم الحقيقي فوضوي. تصوّر نظامك على أنه تجميعة من الشفرة المثالية، وسطوح بينية من الشفرة المثالية إلى العالم الحقيقي الفوضوي، والعالم الحقيقي الفوضوي.

خلال عملك في النظام، تستطيع أن تبدأ بنقل الشفرة عبر "واجهة العالم الحقيقي" إلى عالم مثالي أكثر تنظيماً. عندما تبدأ بالعمل في أنظمة تراثية، فإن الشفرة التراثية المكتوبة بفقر قد تشكّل تقريباً كل النظام. أحد السياسات التي تعمل هنا بشكل جيد هي: في أي وقت تلامس فيه قسم من الشفرة الفوضوية، عليك أن ترفعه إلى معايير كتابة الشفرة الحالية، وتعطيه أسماء متحولات واضحة، وهكذا-نقله بشكل فعال إلى العالم المثالي. مع مرور الوقت يستطيع هذا حمل أعباء التحسين السريع في الشفرة الأساس، كما يظهر في الشكل 24-3.



الشكل 24-3 إحدى استراتيجيات تحسين شفرة الإنتاج هي أن تعيد تصنيع الشفرة التراثية المكتوبة بشكل فقير إذا لامستها، بحيث تكون كأنك تنقلها إلى الجانب الآخر من "السطح البيني إلى العالم الحقيقي الفوضوي"

لائحة اختبار: إعادة التصنيع بأمان[1]

- هل كل تغيير هو جزء من استراتيجية تغيير خاضعة للنظام؟
- هل حفظت الشفرة التي بدأت بها قبل أن تبدأ بإعادة التصنيع؟
- هل تحافظ على صغر كل إعادة تصنيع؟
- هل تقوم بإعادات التصنيع واحدة واحدة في المرة الواحدة؟
- هل صنعت لائحة بالخطوات التي تنوي أن تخطوها خلال إعادة التصنيع خاصتك؟
- هل تملك باحة وقوف بحيث تستطيع تذكر الأفكار التي تأتي في منتصف إعادة التصنيع خاصتك؟
- هل قمت بإعادة الاختبار بعد كل إعادة تصنيع؟
- هل تمت مراجعة التغييرات إن كانت هذه التغييرات معقدة أو إن كانت تؤثر بشفرة لمهمة حرجة؟
- هل حسبت حساب خطورة إعادة التصنيع المحددة وعيّرت نهجك تبعاً؟
- هل يحسّن التغيير من جودة البرنامج الداخلية بدلاً من أن يخفضّها؟
- هل تجنبت استخدام إعادة التصنيع كغلاف وإصلاح للشفرة أو كحجة كي لا تعيد كتابة شفرة سيئة؟

مصادر إضافية[2]

تتشارك عملية إعادة التصنيع بالكثير من الأشياء مع عملية إصلاح العيوب. للمزيد حول إصلاح العيوب، انظر القسم 23.3، "إصلاح عيب." المخاطر المرتبطة بإعادة التصنيع مشابهة للمخاطر المرتبطة بمعايرة الشفرة. للمزيد عن إدارة مهام معايرة الشفرة، انظر القسم 25. 6، "ملخص نهج ضبط الشفرة."

Fowler, Martin. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison Wesley, 1999. هذا دليل نهائي لإعادة التصنيع. أنه يحوي نقاشاً تفصيلياً عن العديد من إعادات التصنيع الخاصة الملخَّصة في هذا الفصل، بالإضافة إلى حفنة من إعادات التصنيع الأخرى غير الملخصة هنا. قدم فولير العديد من الأمثلة بالشفرة ليوضح كيف تُنجز كل إعادة تصنيع خطوة بخطوة.

[1] cc2e.com/2457

[2] cc2e.com/2464

نقاط مفتاحية

- التغيّر في البرنامج هو حقيقة حياتية في كلا التطوير الأولي وبعد الإصدار الأولي.
- يمكن للبرمجية إما أن تتحسن أو تسوء إذا تغيرت. القاعدة الرئيسية لنمو البرمجية هي: ينبغي أن تتحسن الجودة الداخلية مع نمو الشفرة.
- أحد مفاتيح النجاح في إعادة التصنيع أن تتعلم كيف تنتبه إلى إشارات التحذير المتعددة أو الروائح التي تشير إلى الحاجة إلى إعادة التصنيع.
- مفتاح آخر لإعادة تصنيع ناجحة أن تتعلم إعادات التصنيع المخصصة المتعددة.
- مفتاح أخير للنجاح أن تمتلك استراتيجية لتعيد التصنيع بأمان. بعض نُهج إعادة التصنيع أفضل من بعض.
- إعادة التصنيع خلال التطوير هي الفرصة الفضلى التي يمكن أن تحصل عليها لتحسّن برنامجك، وتقوم بكل التغييرات التي ترغب بأن تكون صنعتها من المرة الأولى. استغل هذه الفرص خلال التطوير!

الفصل 25

استراتيجيات ضبط الشفرة

المحتويات1

مواضيع ذات صلة

- تقنيات ضبط الشفرة. الفصل 26.
- هيكلة البرمجيات: القسم 3.5.

يناقش هذا الفصل مسألة ضبط الأداء - تاريخياً، والتي تعتبر قضية مثيرة للجدل. كانت موارد الحاسوب محدودة للغاية في ستينيات القرن الماضي، وكانت الكفاءة من بواعث القلق الرئيسية. ومع ازدياد قوة أجهزة الحواسب في السبعينيات، أدرك المبرمجون كم كان لتركيزهم على الأداء آثاراً سلبية على مواضيع قابلية القراءة والصيانة وضبط الشفرة والتي لقيت اهتماماً أقل. عادت قيود الأداء مجدداً مع ثورة الحواسيب الصغيرة في ثمانينات القرن الماضي جالبةً معها موضوع الكفاءة إلى المقدمة، والتي أخذت بالتضاؤل خلال التسعينات. وفي عام 2000 عادت قيود محدودية الذاكرة في البرمجيات المضمنة بالأجهزة كالهواتف وأجهزة PDA ومحدودية زمن تنفيذ تفسير الشفرة عادت مرة أخرى لتجعل من موضوع الكفاءة موضوعا رئيسيا.

[1] cc2e.com/2578

يمكنك معالجة موضوع الأداء على مستويين: استراتيجي وتقني. يعالج هذا الفصل قضايا الأداء الاستراتيجية: ما هو الأداء، ومدى أهميته، والنُهج العامة لإنجازه. إذا كنت تملك بالفعل معرفة جيدة باستراتيجيات ضبط الشفرة وتبحث عن تقنيات ضبط شفرة محددة تحسّن الأداء، فانتقل إلى الفصل 26 "تقنيات ضبط الشفرة." قبل أن تبدأ أي عمل أداء أساسي، على الأقل، سيساعدك المرور السريع على المعلومات في هذا الفصل في عدم إضاعة الوقت في التحسين عندما يتوجب عليك القيام بأنواع أخرى من العمل.

25.1 نظرة عامة على الأداء

إن ضبط الشفرة هو أحد طرق تحسين أداء البرنامج. ويمكنك غالباً إيجاد طرق أخرى لتحسين أداءه –بوقت أقل وضرر أقل بالشفرة- من استخدام ضبط الشفرة. يصف هذا القسم هذه الخيارات.

خصائص الجودة والأداء

ينظر بعض الناس إلى العالم من خلال نظارات ذات ألوان زاهية. يميل المبرمجون مثلك ومثلي إلى النظر إلى العالم عبر خلال نظارات شفرة ملونة. مفترضين أنه كلما كانت شفرتنا أفضل، فإن زبائننا وعملائنا سيحبون برامجنا أكثر[1].

قد يكون لوجه النظر هذه عنوان بريدي في مكان ما من الواقع، ولكن ليس لها رقم شارع ولا تمتلك بالتأكيد أية ممتلكات. يهتم المستخدمون بخصائص البرنامج الملموسة أكثر من اهتمامهم بجودة الشفرة البرمجية. في بعض الأحيان يهتم المستخدمون بالأداء الخام، ولكن فقط عندما يؤثر ذلك على عملهم. إذ يميل المستخدمون إلى أن يكونوا أكثر اهتماما بإنتاجية البرنامج من الأداء الخام. غالبًا ما يكون استلام البرامج في الوقت المحدد وتوفير واجهة مستخدم واضحة وتجنب أوقات التوقف عن العمل أكثر أهمية بالنسبة لهم.

وهنا توضيح. التقطُ على الأقل 50 صورة في أسبوع بواسطة الكاميرة الرقمية الخاصة بي. ولرفع الصور إلى حاسبي، تتطلب البرمجية المرفقة بالكاميرة مني تحديد الصور صورة صورة، واستعراضهم في نافذة تعرض فقط 6 صور في كل مرة. إن تحميل 50 صورة عملية متعبة وتتطلب دزينات من نقرات الفأرة والكثير من التنقلات خلال نافذة الصور الستة. بعد تعاملي بهذه الطريقة لعدة أشهر، اشتريت قارئ لبطاقة الذاكرة يمكن توصيله بشكل مباشر بحاسبي بحيث يعتبرها الحاسب كمحرك أقراص. الآن يمكنني استخدام مستعرض نظام التشغيل `Windows Explorer` لنسخ الصور إلى الحاسب الخاص بي. وما كان يتطلب عشرات نقرات الماوس والكثير من الانتظار الآن يمكن اختصاره بنقرتين، `Ctrl + A`، والسحب والافلات. لا يهمني حقا ما إذا كان قارئ

[1] الكثير من الأخطاء ارتكبت باسم الكفاءة (دون إنجازها بالضرورة) أكثر من أي سبب آخر بما في ذلك الغباء الأعمى- W. A. Wulf

بطاقة الذاكرة ينقل كل ملف أو كل ملفين في المرة كما في البرمجية السابقة، لأن سرعة نقل البيانات الخاصة بي أسرع. بصرف النظر عما إذا كانت شفرة قارئ بطاقة الذاكرة أسرع أو أبطأ، إلاّ أن الأداء أفضل.

نقطة مفتاحية

إن الأداء مرتبط فقط بشكل غير محكم بسرعة الشفرة. بقدر ما تعمل على سرعة شفرتك، فإنك لا تعمل على خصائص الجودة الأخرى. كن حذرا من التضحية بخصائص أخرى فقط لجعل التعليمات البرمجية الخاصة بك أسرع. فقد يضر عملك على السرعة بالأداء العام بدلاً من مساعدته.

الأداء وضبط الشفرة

بعد اختيارك للكفاءة كأولوية، سواء أكان التركيز على السرعة أو الحجم، يجب مراعاة عدة خيارات قبل اختيار تحسين السرعة أو الحجم على مستوى الشفرة. فكر في الكفاءة من كل من وجهات النظر هذه:

- متطلبات البرنامج
- تصميم البرنامج
- تصميم الصف والإجرائية
- التفاعلات مع نظام التشغيل
- تنسيق الشفرة
- المكونات المادية Hardware
- ضبط الشفرة

متطلبات البرنامج

غالباً ما يتم تحديد الأداء كمتطلب أكثر بكثير مما هو مطلوب في الواقع. يخبر باري بويم قصة نظام في TRW الذي يتطلب وقت تهيئة استجابة بجزء من الثانية. أدى هذا الشرط إلى تصميم معقد للغاية وتكلفة تقديرية بـ 100 مليون دولار. حدد التحليل الإضافي أن المستخدمين سيكونون راضيين عن زمن استجابة أربع ثوان في 90٪ من الحالات. تعديل متطلبات زمن الاستجابة تخفّض تكلفة النظام الكلية حوالي 70 مليون دولار (Boehm 2000b).

قبل أن تستثمر الوقت في حل مشكلة الأداء، تأكد من حل المشكلة التي تحتاج فعلا إلى حل.

تصميم البرنامج

يتضمن تصميم البرنامج الخطوط الرئيسية للتصميم لبرنامج واحد، وبالأخص الطريقة التي يتم بها تقسيم البرنامج إلى صفوف[1]. تجعل بعض تصميمات البرامج من كتابة نظام عالي الأداء أمراً صعباً. في حين أن البعض الآخر يجعل الصعوبة في عدم القيام بذلك.

لنأخذ مثالاً على برنامج جمع البيانات في العالم الحقيقي والذي حدّد تصميمه ذو المستوى العالي، معدل نقل السعة كسمة رئيسية للمنتج. تَضمَّن كل قياس وقتاً لعمل قياس كهربائي، ومعايرة القيمة، وقياس القيمة، وتحويلها من وحدات بيانات المستشعرات (مثل الميلي فولت) إلى وحدات بيانات هندسية (مثل درجات مئوية).

في هذه الحالة، من دون معالجة الخطر في المستوى التصميم العالي، كان المبرمجون سيجدون أنفسهم يحاولون موائمة الرياضيات لتقييم كثير حدود من الدرجة 13في البرمجية - أي، كثير الحدود ب 14 حد، بما في ذلك المتغيرات التي يتم رفعها إلى القوة 13. بدلاً من ذلك، عالجوا المشكلة مع أجهزة مختلفة وتصميم عالي المستوى يستخدم العشرات من كثيرات الحدود من الدرجة الثالثة. هذا التغيير لا يمكن إجراءه من خلال ضبط الشفرة، ومن غير المحتمل أن يكون هناك أي كمية من ضبط الشفرة من الممكن أن تحل المشكلة. هذا مثال لمشكلة يجب معالجتها على مستوى تصميم البرنامج.

إذا كنت تعلم أن حجم البرنامج وسرعته مهمان، فصمم هيكلة البرنامج بحيث تمكنك من تلبية أهداف الحجم والسرعة بشكل مناسب[2]. صمم هيكلة أداء كائنية التوجه، ثم حدّد أهداف الموارد للنظم الفرعية الفردية والميزات والصفوف. سيساعد ذلك بعدة طرق:

- تحديد الأهداف الفردية للموارد يجعل الأداء النهائي للنظام قابلاً للتنبؤ به. إذا حققت كل ميزة أهداف مواردها، فسـوف يحقق النظام بأكمله أهدافه. يمكنك تحديد الأنظمة الفرعية التي تواجه مشـكلة في تحقيق أهدافها مبكرا واستهدافها لإعادة تصميمها أو ضبط شفرتها.
- إن مجرد عمل أهداف واضـحة يحسـن من احتمال تحقيقها. يعمل المبرمجون لتحقيق الأهداف عندما يعرفون ما هي؛ كلما كانت الأهداف أكثر وضوحا، كلما كان العمل على تحقيقها أسهل.
- 

يمكنك تحديد الأهداف التي لا تحقق الكفاءة بشكل مباشر ولكن تعزز الكفاءة على المدى الطويل. غالباً ما يتم الوصول إلى أفضل كفاءة في سياق القضايا الأخرى. على سبيل المثال، يمكن لتحقيق درجة عالية من قابلية التعديل أن يوفر أساسًا أفضل لتحقيق أهداف الكفاءة بدلاً من تحديد هدف الكفاءة

[1] **إشارة مرجعية** لتفاصيل عن تصميم الأداء في البرنامج، انظر قسم "موارد إضافية" في نهاية هذا الفصل.

[2] **إشارة مرجعية** لتفاصيل عن الطريقة التي يعمل بها المبرمجون لتحقيق الأهداف، انظر "تحديد الأهداف" في القسم 20.2

بشكل صريح. بفضل التصميم المعياري القابل للتعديل، يمكنك بسهولة تبديل المكونات الأقل كفاءة بمكونات أكثر كفاءة.

تصميم الصف والإجرائية

تصميم الأجزاء الداخلية للصف والإجرائيات فرصة أخرى للتصميم من أجل الأداء.[1] يتمثل أحد مفاتيح الأداء التي تدخل في هذا المستوى في اختيار أنواع البيانات والخوارزميات، والتي تؤثر عادةً على كل من استخدام ذاكرة البرنامج وسرعة التنفيذ. هذا هو المستوى الذي يمكنك عنده اختيار خوارزمية الفرز السريع `quicksort` بدلاً من خوارزمية الفرز الفقاعي `bubblesort` أو البحث الثنائي بدلاً من البحث الخطي.

التفاعلات مع نظام التشغيل

إذا كان برنامجك يعمل مع ملفات خارجية أو ذاكرة ديناميكية أو أجهزة خرج، فمن المحتمل أنه يتفاعل مع نظام التشغيل.[2] إذا لم يكن الأداء جيدًا، فقد يرجع ذلك إلى أن إجرائيات نظام التشغيل بطيئة أو ثقيلة. قد لا تكون على دراية بأن البرنامج يتفاعل مع نظام التشغيل؛ في بعض الأحيان، يقوم المترجم بإنشاء استدعاءات للنظام أو تقوم المكتبات التي تستخدمها بالاعتماد على استدعاءات نظام لا تتوقعها مطلقا. يوجد المزيد عن هذا في الأقسام التالية.

تنسيق الشفرة

تحوّل المترجمات الجيدة شفرة اللغة عالية المستوى الواضحة إلى شفرة الآلة الموائمة. إذا اخترت المترجم الصحيح، فقد لا تحتاج إلى التفكير في تحسين السرعة أكثر من ذلك.

توفر نتائج التحسين المذكورة في الفصل 26 أمثلة عديدة عن تحسينات المترجم التي تنتج شفرة أكثر كفاءة من عملية ضبط الشفرة يدويا.

المكونات المادية `Hardware`

في بعض الأحيان، تكون الطريقة الأرخص والأفضل لتحسين أداء البرنامج هي شراء عتاد جديد. إذا كنت بصدد توزيع برنامج للاستخدام على مستوى البلد من قبل مئات الآلاف من العملاء، فإن شراء أجهزة جديدة ليس خياراً عملياً. أما إذا كنت تطور برامج مخصصة لعدد قليل من المستخدمين المحليين، فقد تكون ترقية الأجهزة

[1] **إشارة مرجعية** لمزيد من المعلومات حول أنواع البيانات والخوارزميات، انظر قسم الـ " الموارد الإضافية" في نهاية الفصل.

[2] **إشارة مرجعية** لمعلومات عن الاستراتيجيات على مستوى الشفرة التي تعالج إجرائيات نظام التشغيل البطيئة، انظر الفصل 26"تقنيات ضبط الشفرة"

الخيار الأرخص. فهي توفر من كلفة العمل الأولية للأداء. كما توفر من كلفة مشاكل الصيانة المستقبلية الناجمة عن عمل التحسين. كما تحسن أداء كل البرامج الأخرى التي تعمل على هذا الجهاز أيضا.

ضبط الشفرة

ضبط الشفرة هو عملية تعديل الشفرة الصحيحة بطرق تجعلها تعمل بشكل أكثر كفاءة، وهو موضوع بقية هذا الفصل. "ضبط" يشير إلى تغييرات صغيرة النطاق والتي تؤثر على صف واحد أو إجرائية واحدة أو، بشكل أكثر شعبيةً، بضعة أسطر من الشفرة. لا يشير "الضبط" إلى تغييرات التصميم على نطاق واسع أو غيرها من المستويات الأعلى لتحسين الأداء.

يمكنك إجراء تحسينات دراماتيكية على كل مستوى من تصميم النظام من خلال ضبط الشفرة. يستشهد جون بنتلي بفرضية أنه في بعض الأنظمة يمكن مضاعفة التحسينات في كل مستوى (1982). ولأنه بإمكانك إنجاز بمقدار عشرة أضعاف في كل مستوى من المستويات الستة، فهذا يعني تحسينًا محتملًا للأداء بمقدار مليون مرة. على الرغم من أن مثل هذا المضاعفة من التحسينات تتطلب برنامجًا تكون فيه المكاسب عند مستوى واحد مستقلة عن المكاسب في المستويات الأخرى، وهو أمر نادر الحدوث، إلا أنّ الإمكانية محفّزة.

25.2 مقدمة إلى ضبط الشفرة

ما هو سر جاذبية ضبط الشفرة؟ فهي ليست الطريقة الأكثر فاعلية لتحسين الأداء - فهيكلة البرامج، وتصميم الصفوف، واختيار الخوارزمية عادة ما يؤدي إلى تحسينات ملحوظة أكثر. كما أنه ليس أسهل طريقة لتحسين الأداء - فشراء أجهزة جديدة أو مترجم مع محسّن أفضل يعتبر أسهل. وليس بالطريقة الأرخص لتحسين الأداء – إذ يستغرق الأمر وقتًا أطول للمعالجة التحضيرية لضبط الشفرة، ومن الصعب لاحقاً الحفاظ على الشفرة المضبوطة يدوياً.

ضبط الشفرة جذّاب لعدة أسباب. أحد الأسباب هو أنه يبدو كأنه يتحدى قوانين الطبيعة. فمن الممتع بشكل لا يصدق أن تأخذ إجرائية تنفذ في 20 ميكرو ثانية، وتضبط بضعة أسطر فيها، فتقلل سرعة التنفيذ إلى 2 ميكرو ثانية.

كما أنها جذابة لأن إتقان فن كتابة الشفرة الفعّالة هو من المراحل التي تمر بها لتصبح مبرمجًا قويا. في التنس، الطريقة التي تلتقط فيها كرة تنس لا تعطيك أي نقاط لعب إضافية، ولكنك ستكون بحاجة إلى معرفة الطريقة الصحيحة للقيام بذلك. لا يمكنك فقط أن تميل وتلتقطها بيدك. إذا كنت جيدًا، فقم بضربها بقوة برأس مضربك حتى ترتد مرتفعة إلى مستوى خصرك ثم تلتقطها. ضربها أكثر من ثلاث مرات، وأكثر من ذلك إن لم ترتد من المرة الأولى، هو فشل ذريع. على الرغم من أنها تبدو غير مهمة، إلا أن الطريقة التي تلتقط بها الكرة تحمل طابعًا

معينًا داخل ثقافة لعبة التنس. وبشكل مشابه، لا يوجد عادة من يهتم بمدى متانة شفرتك غيرك أنت والمبرمجين الآخرين. ومع ذلك، ففي إطار ثقافة البرمجة، فإن كتابة شفرة مستوى الوحدات الصغيرة تثبت أنك ممتاز.

تكمن المشكلة في ضبط الشفرة في أن الشفرة الفعالة ليست بالضرورة شفرة "أفضل". وهو موضوع الأقسام القليلة القادمة.

مبدأ باريتو The Pareto Principle

ينص مبدأ باريتو، المعروف أيضًا بقاعدة 20/80، على أنه بإمكانك الحصول على 80 بالمئة من النتيجة مع 20 في المئة من المجهود. ينطبق المبدأ في الكثير من المجالات الأخرى من البرمجة، وهو يُطبَّق بالتأكيد على تحسين البرنامج.

نقطة مفتاحية

أفاد باري بويم Barry Boehm أن 20 في المئة من إجراءات البرنامج تستهلك 80 في المئة من زمن تنفيذه (1987 ب). وفي بحثه التقليدي "دراسة تجريبية لبرامج " فورتران" Fortran "، وجد دونالد كنوث Donald Knuth أن أقل من 4 في المئة من البرنامج عادة ما تكون مسؤولة عن أكثر من 50 في المئة من الوقت أثناء عمله (1971).

استخدم Knuth مُوَصِّف لعداد خط إنتاج لاكتشاف هذه العلاقة المدهشة، كما أن التأثيرات على التحسين واضحة. يجب عليك قياس الشفرة البرمجية للعثور على النقاط الفعالة ثم وضع مواردك في تحسين النسبة المئوية القليلة التي تستخدم الجزء الأكبر. وصّف Knuth برنامج عداد لخط إنتاج ووجد أنه يصرف نصف وقت التنفيذ في حلقتين. قام بتغيير بضعة أسطر من الشفرة البرمجية وضاعف بذلك سرعة الموصّف في أقل من ساعة.

يصف جون بينتلي Jon Bentley الحالة التي قضى فيها برنامج مؤلف من 1000 سطر 80 في المئة من وقته في إجرائية الجذر التربيعي المكونة من خمسة أسطر. ومن خلال مضاعفة سرعة إجرائية الجذر التربيعي، تضاعفت سرعة البرنامج (1988).

إن مبدأ باريتو هو أيضًا مصدر لنصيحة كتابة معظم الشفرة بلغة مفسرة مثل بايثون ثم إعادة كتابة النقاط الفعالة بلغة مجمعة أسرع مثل سي.

قدّم بنتلي Bentley أيضاً تقارير عن حالة فريق اكتشف أن نصف وقت نظام التشغيل يتم إنفاقه في حلقة صغيرة. فأعادوا كتابة الحلقة باستخدام **شفرة الميكرو** وجعلوا بذلك الحلقة أسرع بـ 10 مرات، لكن ذلك لم يغير أداء النظام - فهم أعادوا كتابة حلقة خمول النظام!

توصّل الفريق الذي صمم لغة ALGOL - وهي بمنزلة جدّ اللغات الحديثة وأحد أكثر اللغات تأثيرًا على الإطلاق - النصيحة التالية:

"الأفضل هو عدو الجيد." العمل نحو الكمال قد يمنع الإنجاز. اصنعها أولاً، ثم أتقنها. فالجزء الذي يحتاج إلى الكمال يكون صغيرًا عادة.

حكايات الزوجات القديمة[1]

الكثير مما سمعته عن ضبط الشفرة خاطئ، بما في ذلك الالتباس العام التالي:

يحسّن تقليل أسطر التعليمات البرمجية بلغة عالية المستوى من سرعة أو حجم شفرة الآلة الناتجة- خطأ!

العديد من المبرمجين يتشبثون بعناد بالمعتقد بأنهم إذا تمكنوا من كتابة الشفرة في سطر واحد أو سطرين، فسيكون ذلك الأكثر فعالية. خذ بعين الاعتبار التعليمة البرمجية التالية التي تهيئة 10 عناصر لمصفوفة:

```
for i = 1 to 10
a[ i ] = i
end for
```

هل تعتقد أن هذه الأسطر أسرع أو أبطأ من السطور الـ 10 التالية التي تؤدي نفس الوظيفة؟

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

إذا اتبعت الاعتقاد القديم "الأسطر القليلة أسرع"، ستخمن أن أول شفرة هي الأسرع. ولكن الاختبارات في ميكروسوفت فيجوال بيسك وجافا أظهرت أن الجزء الثاني أسرع بنسبة 60٪ على الأقل من الأول. هنا الأرقام:

اللغة	زمن حلقة for	زمن الشفرة الخطية	زمن الحفظ	تقييم الأداء
فيجوال بيسك	8.47	3.16	63%	2.5:1
جافا	12.6	3.23	74%	4:1

ملاحظة (1) يتم إعطاء الأزمنة في هذا الجدول والجداول التالية في هذا الفصل بالثواني وهي ذات معنى فقط للمقارنات بين السطور في كل جدول. تختلف الأوقات الفعلية باختلاف المترجم وخيارات المترجم المستخدمة والبيئة التي يتم فيها تشغيل كل اختبار. (2) عادةً ما تتكون النتائج القياسية من عدة آلاف إلى عدة ملايين من عمليات تنفيذ أجزاء الشفرة لتخفيف التقلبات من عينة إلى عينة في النتائج. (3) لا يتم الإشارة إلى أصناف وإصدارات محددة للمترجم. حيث تختلف خصائص الأداء بشكل كبير من صنف إلى آخر ومن إصدار إلى آخر. (4) لا تكون المقارنات بين النتائج من اللغات المختلفة دائمًا ذات معنى؛ لأن مترجمات للغات المختلفة لا تقدم دائما خيارات قابلة للمقارنة في إنشاء الشفرة. (5) تعتمد النتائج الموضحة للغات

[1] **ترجمة** (old wives' tales) مصطلح يشير إلى المعتقدات القديمة التي أُثبت أنها منافية للعلم.

المفسرة (بي آش بي وبايثون) على أقل من 1٪ من عمليات الاختبار المستخدمة للغات الأخرى. (6) قد لا يمكن إعادة إنتاج بعض نسب "توفير الوقت" بالضبط من البيانات الموجودة في هذه الجداول بسبب تقريب إدخالات "الزمن المباشر" و"زمن الشفرة المضبوطة".

لا يعني هذا بالتأكيد أن زيادة عدد أسطر شفرة اللغة عالية المستوى يؤدي دائمًا إلى تحسين السرعة أو تقليل الحجم. وهذا يعني أنه بغض النظر عن الناحية الجمالية لكتابة شيء ما بأقل عدد ممكن من أسطر الشفرة، فلا توجد علاقة يمكن التنبؤ بها بين عدد أسطر الشفرة بلغة عالية المستوى وبين حجم وسرعة البرنامج.

عمليات معينة على الأرجح ستكون أسرع أو أصغر من غيرها - خطأ! لا يوجد احتمال لوجود "على الأرجح" عندما تتحدث عن الأداء. يجب عليك دائمًا قياس الأداء لمعرفة ما إذا كانت التغييرات التي أجريتها قد ساعدت أو ألحقت الضرر ببرنامجك. تتغير قواعد اللعبة في كل مرة تقوم فيها بتغيير اللغات، أو المترجمات، أو إصدارات المترجمات، أو المكتبات، أو إصدارات المكتبات، أو المعالج، أو مقدار الذاكرة على الجهاز، أو لون القميص الذي ترتديه (حسناً، ليس هذا)، وهلم جرا. ما كان صحيحًا على جهاز محدد بمجموعة محددة من الأدوات يمكن أن يكون خاطئًا على جهاز آخر باستخدام مجموعة مختلفة من الأدوات.

تقترح هذه الظاهرة عدة أسباب لعدم تحسين الأداء باستخدام ضبط الشفرة البرمجية. إذا كنت تريد أن يكون برنامجك محمولاً portable، فإن التقنيات التي تحسن الأداء في بيئة معينة يمكن أن تؤدي إلى تدهورها في بيئة أخرى. إذا قمت بتغيير المترجم أو ترقيته، فقد يقوم المترجم الجديد تلقائيًا بتحسين الشفرة بالطريقة التي تم ضبطها يدويًا وسيضيع عملك.

والأسوأ من ذلك، أنه قد يؤدي ضبط الشفرة الخاص بك إلى إبطال تحسينات مترجم أكثر قوة تم تصميمها للعمل مع شفرة صريحة. عند ضبط الشفرة، فإنك تشترك ضمنيًا في إعادة تهيئة كل عملية تحسين في كل مرة تقوم فيها بتغيير العلامة التجارية لمترجم، وإصدار المترجم، وإصدار المكتبة، وما إلى ذلك.

في حالة عدم إعادة التوصيف `reprofile`، قد يؤدي التحسين الذي يحسن الأداء ضمن إصدار معين لمترجم أو لمكتبة إلى خفض مستوى الأداء عندما تغير بيئة البناء.

يجب عليك التحسين وأنت تسير قدماً— خطأ![1] إحدى النظريات هي أنه إذا كنت تسعى إلى أسرع وأصغر شفرة ممكنة أثناء كتابة كل إجرائية، فسيكون برنامجك سريعًا وصغيراً.

يؤدي هذا النهج إلى إنشاء حالة "الغابة من أجل الأشجار" حيث يتجاهل المبرمجون عمليات تحسين شاملة مهمة لكثرة انشغالهم بالتحسينات الصغيرة. فيما يلي المشاكل الرئيسية للتحسين أثناء المتابعة:

---

[1] يجب أن نتغاضى عن الكفاءات الصغيرة، ففي حوالي 97 ٪ من الحالات: يكون التحسين المبكر هو أصل كل المشاكل. —دونالد كنوث.

- من المستحيل تقريبًا تحديد اختناقات الأداء قبل أن يعمل البرنامج بشكل كامل. فالمبرمجون سيئون للغاية في التخمين أي أربعة في المئة من الشفرة تمثل 50 في المئة من وقت التنفيذ، وبالتالي فإن المبرمجين الذين يقومون بالتحسين أثناء التنقل، سينفقون في المتوسط 96 في المئة من وقتهم في تحسين الشفرة التي لا تحتاج إلى تحسين. وهذا ما لا يترك سوى القليل من الوقت لتحسين الأربعة في المئة التي تهم فعلا.
- في الحالة النادرة والتي يحدد فيها المطورون الاختناقات بشكل صحيح، فإنهم يبالغون في القضاء على الاختناقات التي حددوها ويسمحون لمناطق أخرى بأن تصبح حرجة. مرة أخرى، فإن ناتج التأثير النهائي هو انخفاض في الأداء. يمكن للتحسينات التي تتم بعد اكتمال النظام تحديد مكان كل مشكلة وأهميتها النسبية بحيث يتم تخصيص وقت التحسين بفعالية.
- التركيز على التحسين خلال التطوير الأولي ينتقص من تحقيق أهداف البرنامج الأخرى. ينشغل المطورون في تحليل الخوارزميات والنقاشات الغامضة التي لا تساهم في النهاية بقيمة كبيرة للمستخدم. تصبح المخاوف مثل الصحة، وإخفاء المعلومات، وقابلية القراءة أهدافًا ثانوية، على الرغم من أنه من الأسهل تحسين الأداء بعد هذه المخاوف. فإنه عادةً ما يؤثر العمل اللاحق المتعلق بصنع هذا الأداء على أقل من خمسة في المئة من شفرة البرنامج. هل تفضل العودة والعمل على الأداء على 5 في المئة من الشفرة أو العمل على إمكانية القراءة على 100 في المئة منها؟

باختصار، العقبة الأساسية للتحسين المبكر هي في افتقاره للمنظور.

تتضمن ضحاياه سرعة الشفرة النهائية، وخصائص الأداء الأكثر أهمية من سرعة الشفرة، وجودة البرنامج، وفي النهاية مستخدمي البرنامج. إذا كان وقت التطوير الذي يتم توفيره من خلال تنفيذ البرنامج الأبسط مخصصًا لتحسين البرنامج قيد التنفيذ، فستكون النتيجة دائمًا برنامجًا يتم تشغيله أسرع من برنامج تم تطويره بجهود تحسين عشوائية (Stevens 1981).

أحيانا، لن يكون إجراء هذا التحسين اللاحق كافيًا لتحقيق أهداف الأداء وسيتوجب عليك إجراء تغييرات كبيرة في الشفرة المكتملة. في حالات كهذه، لا تكون التحسينات الصغيرة المحلية قد حققت المكاسب المطلوبة على أي حال.

ليست المشكلة في مثل هذه الحالات عدم كفاية كفاءة الشفرة – ولكن في أن هيكلة البرمجية غير وافية.

إذا كنت بحاجة إلى التحسين قبل اكتمال أحد البرامج، فقم بتقليل المخاطر من خلال بناء المنظور في العملية الخاصة بك. تتمثل إحدى الطرق في القيام بتحديد أهداف الحجم والسرعة للميزات ثم تحسينها لتحقيق الأهداف أثناء العمل. تحديد مثل هذه الأهداف في المواصفة هو طريقة لإبقاء نظرك على "الغابة بينما تكتشف كم هي شجرتك الخاصة كبيرة الحجم.

إن البرنامج السريع لا يقل أهمية عن البرنامج الصحيح – خطأ![1] إنه من الصعب جداً أن يكون صحيحا أن البرامج تحتاج لأن تكون سريعة أو صغيرة قبل حاجتها لأن تكون صحيحة. يروي جيرالد واينبرغ Gerald Weinberg قصة المبرمج الذي تم نقله إلى ديترويت Detroit للمساعدة في تصحيح برنامج مضطرب. عمل المبرمج مع الفريق الذي طور البرنامج وخلص بعد عدة أيام إلى أن الوضع كان ميؤوسًا منه.

وفي الطائرة خلال رحلة عودته إلى الوطن، تفكر مليّا في الوضع وأدرك طبيعة المشكلة. وبنهاية الرحلة، كان قد أصبح لديه مخطط للشفرة الجديدة. قام باختبار الشفرة لعدة أيام وكان على وشك العودة إلى ديترويت عندما وصلته برقية تقول بأن المشروع قد ألغي لأنه كان من المستحيل كتابة البرنامج. على أي حال عاد مجددا الى ديترويت واقنع المسؤولين التنفيذيين بأن المشروع يمكن أن يكتمل.

ثم اضطر إلى إقناع المبرمجين الأصليين للمشروع. استمعوا إلى عرضه التقديمي، وعندما انتهى، سأل منشئ النظام القديم: "وكم يستغرق برنامجك؟"

"هذا يختلف، ولكن حوالي عشر ثوان لكل مدخل."

"آها! ولكن يستغرق برنامجي ثانية واحدة فقط لكل إدخال. "

استند المبرمج القديم إلى الوراء، راضياً عن أرباك المبرمج الجديد. يبدو أن المبرمجين الآخرين موافقون، لكن المبرمج الجديد لم يكن خائفا.

"نعم، لكن برنامجك لا يعمل. إذا توقف برنامجي عن العمل، فيمكنني تشغيله حالاً."

بالنسبة لشريحة معينة من المشاريع، تعتبر السرعة أو الحجم مصدر قلق كبير. هذه الشريحة هي أقليّة، وهي أصغر بكثير مما يعتقده معظم الناس وتصبح أصغر يوما بعد يوم.

بالنسبة لهذه المشاريع، يجب معالجة مخاطر الأداء بالتصميم المسبق. أما بالنسبة للمشاريع الأخرى، فيشكل التحسين المبكر تهديدًا كبيرًا لجودة البرامج العامة، بما في ذلك الأداء.

متى نقوم بالضبط "tune "

[1] **مزيد من القراءة** للعديد من الحكايات المسلية والمفيدة الأخرى، انظر *علم نفس برمجة الحاسوب* جيرالد واينبرغ (1998).

استخدم تصميمًا عالي الكفاءة.[1] اجعل البرنامج صحيحا. اجعله معياريا ويمكن تعديله بسهولة بحيث يسهل العمل عليه لاحقًا. عندما يصبح كاملا وصحيحا تحقق من الأداء. إذا كان برنامجا بطيئا اجعله سريعا وصغيرا. لا تقم بالتحسين حتى تتأكد بأنك بحاجة لذلك.

منذ عدة سنوات، عملت في مشروع سي ++ ينتج مخرجات رسومية لتحليل بيانات الاستثمار. بعد أن حصل فريقي على أول رسم بياني، أفاد الاختبار بأن البرنامج استغرق حوالي 45 دقيقة لرسم المخطط البياني، وهو أمر غير مقبول بشكل واضح.

عقدنا اجتماع للفريق لتحديد ما يتوجب علينا القيام به حيال ذلك. عندها بدا أحد المطورين غاضباً وصاح قائلاً: "إذا كنا نرغب في الحصول على أي فرصة لإطلاق منتج مقبول، علينا أن نبدأ في إعادة كتابة كامل الشفرة الأساسية في المجمّع الآن"

أجبت بأنني لم أكن أعتقد ذلك - إن أربعة بالمئة من الشفرة ربما مسؤولة عن 50 بالمئة أو أكثر من عنق الزجاجة "الاختناقات" في الأداء. وسيكون من الأفضل معالجة أربعة في المئة في نهاية المشروع.

بعد قليل من الصراخ، عيّنني مديرنا للقيام ببعض أعمال الأداء الأولية (التي كانت حقاً حالة من "أوه لا! من فضلك لا ترمي بي في مستنقع الشوك هذا!")

وكما هو الحال غالباً، أظهر يوم عمل زوجاً من الاختناقات الواضحة في الشفرة. قلص عدد قليل من تغييرات ضبط الشفرة وقت الرسم من 45 دقيقة إلى أقل من 30 ثانية. كان أقل من واحد بالمئة من الشفرة مسؤولاً عن 90 بالمئة من زمن التشغيل. وبحلول وقت إصدارنا للبرنامج بعد عدة أشهر، خفضت عدة تغييرات إضافية لضبط الشفرة ذلك الزمن إلى أكثر من ثانية واحدة بقليل.

تحسينات المترجم

قد تكون تحسينات المترجمات الجديدة أقوى مما تتوقع. في الحالة التي أوضحتها سابقاً، قام المترجم الخاص بي بعمل جيد لتحسين حلقة "معششة"، حيث كنت قادراً على القيام بذلك عن طريق إعادة كتابة الشفرة بأسلوب يفترض أنه أكثر كفاءة. عند شرائك لمترجم، قارن أداء كل مترجم على برنامجك. فكل مترجم لديه نقاط قوة ونقاط ضعف مختلفة، وبعضها سيكون أكثر ملاءمة لبرنامجك من البعض الآخر.

---

[1] **قواعد جاكسون للتحسين**: القاعدة 1. لا تفعل ذلك/أي لا تقم بالتحسين.

القاعدة 2 (للخبراء فقط). لا تفعل ذلك أيضاً- أي حتى يكون لديك حل واضح تماما وغير تفاؤلي.

M. A. Jackson-

يعد تحسين المترجمات أفضل في تحسين الشفرة المباشرة مما هو عليه في تحسين الشفرة الذكية. إذا كنت تقوم بأشياء "ذكية" مثل العبث مع الفهارس الحلزونية، فإن المترجم سيواجه وقت صعب للقيام بعمله وسيؤثر بالتالي على برنامجك.

راجع "استخدام عبارة واحدة فقط لكل سطر" في القسم 31. 5 على سبيل المثال، النتيجة في النهج المباشر في الشفرة المحسنة من قبل المترجم كانت أسرع ب 11 بالمئة مقارنة بالشفرة الذكية.

مع تحسين جيد للمترجم، يمكن أن تتحسن سرعة شفرتك بنسبة 40 بالمئة أو أكثر متضمناً كل شيء. تنتج العديد من التقنيات الموضحة في الفصل التالي مكاسب بنسبة 15-30 بالمئة فقط. لماذا لا تكتب فقط شفرة واضحة وتدع المترجم يقوم بالعمل؟ فيما يلي نتائج بعض الاختبارات للتحقق من سرعة المحسن في إجرائية إدراج-فرز:

اللغة	الزمن من دون تحسينات المترجم	الزمن مع تحسينات المترجم	توفير الوقت	نسبة الأداء
مترجم سي++ 1	2.21	1.05	52%	2:1
مترجم سي++ 2	2.78	1.15	59%	2.5:1
مترجم سي++ 3	2.43	1.25	49%	2:1
مترجم سي#	1.55	1.55	0%	1:1
فيجوال بيسك	1.78	1.78	0%	1:1
جافا في ام 1	2.77	2.77	0%	1:1
جافا في ام 2	1.39	1.38	أقل من 1%	1:1
جافا في ام 3	2.63	2.63	0%	1:1

كان الفرق الوحيد بين إصدارات الإجرائية هو في أنه تم إيقاف تشغيل تحسينات المترجم في أول ترجمة وتم تشغيلها في المرة الثانية. من الواضح أن بعض المترجمات أفضل من البعض الآخر، وبعضهم بدون تحسينات أفضل من البعض الآخر. بعض آلات جافا الافتراضية (JVMs) هي أيضا أفضل من بعضها الأخر بوضوح.

سيكون عليك التحقق من مترجمك الخاص، أو آلة جافا الافتراضية JVM، أو كليهما لقياس التأثير.

25. 3 أنواع السمن والدبس

تجد خلال ضبط الشفرة أجزاء من البرنامج لزجة مثل "الدبس" في فصل الشتاء وضخمة مثل" ديناصور ضخم" Godzilla وتغيّرها لتصبح سريعة ونحيفة حتى أنها يمكن أن تختبئ في المساحات بين البايتات الأخرى في

ذاكرة الوصول العشوائي. يجب عليك دائمًا "توصيف" البرنامج لتعرف بوضوح أي الأجزاء هي بطيئة وسمينة، لكن بعض العمليات لها تاريخ طويل مع الكسل والسمنة، ويمكنك البدء بالتحقيق منها.

المصادر الشائعة لعدم الفعالية

فيما يلي العديد من المصادر الشائعة لعدم الفعالية:

عمليات الإدخال / الإخراج واحدة من أهم مصادر عدم الفعالية هي عمليات الإدخال / الإخراج غير الضرورية (I/O). إذا كان لديك خيار العمل مع ملف في الذاكرة مقابل العمل معه على القرص أو في قاعدة بيانات أو عبر شبكة، استخدم بنية بيانات في الذاكرة ما لم يكن المجال حرجا.

فيما يلي مقارنة أداء بين شفرة تصل إلى عناصر عشوائية في مصفوفة في الذاكرة مكون من 100 عنصر وشفرة تصل إلى عناصر عشوائية من نفس الحجم في ملف قرص يحوي 100 سجل:

اللغة	زمن الملف الخارجي	زمن البيانات في الذاكرة	توفير الوقت	نسبة الأداء
سي++	6.04	0.000	100%	غير متاح
سي #	12.8	0.010	100%	1000:1

وفقًا لهذه البيانات، يكون الوصول إلى الذاكرة أسرع بحدود 1000 مرة من الوصول إلى البيانات في ملف خارجي. بالتأكيد مع مترجم الـسي++ الذي أستخدم، لم يكن الوقت المطلوب للوصول إلى الذاكرة قابلاً للقياس.

تشبه مقارنة الأداء لاختبار مماثل لأزمنة الوصول التسلسلي ما يلي:

اللغة	زمن الملف الخارجي	زمن البيانات في الذاكرة	توفير الوقت	نسبة الأداء
سي++	3.29	0.021	99%	150:1
سي #	2.60	0.030	99%	85:1
ملاحظة: تم تنفيذ اختبارات الوصول التسلسلي مع 13 ضعف من حجم بيانات الاختبارات للوصول العشوائي، لذلك فالنتائج غير قابلة للمقارنة عبر نوعي الاختبارات.				

إذا استخدم الاختبار وسيطًا أبطأ للوصول الخارجي - على سبيل المثال، قرصًا صلبًا عبر اتصال شبكة - فسيكون الفارق أكبر. يشابه هذا الأداء حالة إجراء اختبار وصول عشوائي مماثل على موقع شبكة بدلاً من الجهاز المحلي:

اللغة	زمن الملف المحلي	زمن الملف على الشبكة	توفير الوقت
سي++	6.04	6.64	%10-
سي #	12.8	14.1	%10-

بالطبع، يمكن أن تختلف هذه النتائج بشكل كبير بناءً على سرعة الشبكة الخاصة بك، وتحميل الشبكة، وبعد جهازك المحلي عن محرك الأقراص المتصل بالشبكة، وسرعة محرك الأقراص الشبكة مقارنة بسرعة محرك الأقراص المحلي، والطور الحالي للقمر، وعوامل أخرى.

بشكل عام، يكون تأثير الوصول إلى الذاكرة هامًا بما فيه الكفاية لتجعلك تفكر مرتين بوجود دخل\خرج في جزء حساس للسرعة لبرنامج.

التصحيف Paging عملية تجعل نظام التشغيل يقوم بالتنقل بين صفحات الذاكرة هي أبطأ بكثير من العملية التي تكون على صفحة واحدة فقط من الذاكرة. في بعض الأحيان يُحدث تغيير بسيط فرقًا كبيرًا. في المثال التالي، كتب أحد المبرمجين حلقة التهيئة التي أنتجت العديد من أخطاء الصفحات على نظام يستخدم صفحات K4.

مثال جافا لتهيئة حلقة تسبب الكثير من أخطاء الصفحة

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {
for ( row = 0; row < MAX_ROWS; row++ ) {
table[ row ][ column ] = BlankTableElement();
}
}
```

هذه حلقة منسقة بشكل جيد مع أسماء متغيرات جيدة، فما المشكلة؟ المشكلة هي في أن كل عنصر في الجدول يبلغ طوله 4000 بايت تقريبًا. إذا كان الجدول يحتوي على عدد كبير جدًا من الأسطر "السجلات"، ففي كل مرة يصل فيها البرنامج إلى سطر مختلف، سيتوجب على نظام التشغيل تبديل صفحات الذاكرة. الطريقة التي بنيت فيها الحلقة، هي أنّ كل دخول واحد إلى المصفوفة يبدل أسطر، مما يعني أن كل دخول واحد إلى المصفوفة يسبب التصحيف إلى القرص.

قام المبرمج بإعادة هيكلة الحلقة بهذه الطريقة:

مثال جافا لتهيئة حلقة تسبب أخطاء صفحة قليلة

```
for ( row = 0; row < MAX_ROWS; row++ ) {
for ( column = 0; column < MAX_COLUMNS; column++ ) {
table[ row ][ column ] = BlankTableElement();
}
}
```

لا تزال هذه الشفرة تسبب خطأ صفحة في كل مرة يقوم فيها بتبديل الأسطر، ولكنه يقوم بتبديل الأسطر فقط `MAX_ROWS` مرة بدلاً من `MAX_COLUMNS * MAX_ROWS` مرة.

تختلف ضريبة الأداء المحدد بشكل ملحوظ. قمت بقياس عينة الشفرة الثانية والتي كانت أسرع بنحو 1000 مرة من عينة الشفرة الأولى وذلك على جهاز بذاكرة محدودة. أما على الأجهزة المزودة بذاكرة أكبر، قمت بمعايرة الفرق ليكون صغيرًا بقدر عامل 2، ولم يكن يظهر مطلقاً باستثناء القيم الكبيرة جدًا لـ *MAX_ROWS* و*MAX_COLUMNS*.

استدعاءات النظام غالبًا ما تكون استدعاءات إجرائيات النظام باهظة التكلفة. غالبًا ما تشتمل على حالة تبديل- حفظ حالة البرنامج، واستعادة حالة النواة kernel والعكس. تتضمن إجراءات النظام عمليات الإدخال/الإخراج إلى القرص أو لوحة المفاتيح أو الشاشة أو الطابعة أو أي جهاز آخر؛ وإجرائيات إدارة الذاكرة؛ وبعض الإجرائيات المفيدة.

إذا كان الأداء مهما بالنسبة لك، تعرف على مدى تكلفة استدعاءات نظامك. إذا كانت باهظة التكلفة، ففكر في هذه الخيارات:

- اكتب خدماتك الخاصـــة. في بعض الأحيان تحتاج فقط إلى جزء صــغير من الوظائف التي توفرها إجرائية نظام ويمكن أن تبني إجرائيتك من إجرائيات النظام الأقل مســتوى. كتابة بدائلك الخاصـــة تمنحك شيئًا أسرع وأصغر حجمًا وأكثر ملاءمة لاحتياجاتك.
- تجنب الذهاب إلى النظام.
- اعمل مع بائع النظام لجعل الاســتدعاء أســرع. يرغب معظم البائعين في تحســين منتجاتهم ويســرهم معرفة الأجزاء من أجهزتهم ذات الأداء الضعيف. (قد يبدون متذمرين قليلاً في البداية، لكنهم مهتمون حقاً).

في جهود ضبط الشفرة التي وصفتها في "متى نقوم بالضبط" في القسم 25. 2، استخدم البرنامج صف AppTime المشتق من صف BaseTime المتاح تجاريًا. (تم تغيير هذه الأسماء لحماية المذنبين.) كان الكائن AppTime هو الكائن الأكثر شيوعًا في هذا التطبيق، وقمنا بإنشاء عشرات الآلاف من كائنات AppTime.

بعد عدة أشهر، اكتشفنا أن BaseTime كان يهيئ نفسه لوقت النظام في كل باني. وفي حالتنا، كان وقت النظام غير ذي صلة، مما يعني أننا كنا ننتج بلا داعٍ آلاف الاستدعاءات على مستوى النظام. إن مجرد الاهتمام بباني BaseTime وتهيئة الحقل الزمني إلى القيمة 0 بدلاً من قيمة وقت النظام قد أعطانا الكثير من تحسين الأداء بقدر يوازي التغيرات الأخرى التي أجريناها مجتمعة.

اللغات المفسرة تميل اللغات المفسرة إلى فرض غرامات أداء كبيرة لأنها يجب أن تعالج كل تعليمات لغة البرمجة قبل إنشاء شفرة الآلة وتنفيذها.

في تقييم الأداء الذي أجريته لهذا الفصل والفصل 26، فقد راقبت العلاقات التقريبية في الأداء بين اللغات المختلفة الموضحة في الجدول 25-1.

الجدول 25-1 وقت تنفيذ لغات البرمجة النسبي

اللغة	نوع اللغة	زمن التنفيذ النسبي مقارنة بـسي++
سي++	مترجمة	1:1
فيجوال بيسك	مترجمة	1:1
سي#	مترجمة	1:1
جافا	شفرة بايت	1.5:1
بي إتش بي	مفسّرة	100:1<
بايثون	مفسّرة	100:1<

كما ترون، سي++، فيجوال بيسك، وسي # كلها لغات قابلة للمقارنة. جافا قريبة منها لكنها تميل إلى أن تكون أبطأ من اللغات الأخرى. بي آش بي وبايثون لغات مفسرة، وتميل الشفرة في هذه اللغات إلى أن تكون أبطأ ب 100 مرة أو أكثر من الشفرة في سي++، وفيجوال بيسك، وسي#، وجافا. يجب عرض الأرقام العامة الواردة في هذا الجدول بحذر. بالنسبة لأي جزء معين من الشفرة، سي ++، أو فيجوال بيسك، أو سي #، أو جافا قد تكون أسرع مرتين أو بنصف سرعة اللغات الأخرى. (يمكنك رؤية ذلك بنفسك في الأمثلة التفصيلية في الفصل 26).

الأخطاء: المصدر الأخير لمشاكل الأداء هو الأخطاء في التعليمات البرمجية. يمكن أن تتضمن الأخطاء ترك شفرة تصحيح الأخطاء قيد التشغيل (مثل تسجيل معلومات التتبع إلى ملف)، مع نسيان إلغاء تخصيص الذاكرة، وتصميم جداول قاعدة البيانات بشكل غير صحيح، وتحسس أجهزة غير موجودة حتى ينتهي وقتها، وهكذا.

كانت إصدارة التطبيق 1.0 التي عملت عليها فيها عملية محددة أبطأ بكثير من العمليات المماثلة الأخرى.

نما قدر كبير من أساطير المشروع لشرح سبب بطء هذه العملية. قمنا بإطلاق الإصدار 1.0 بدون فهم كامل لماذا كانت هذه العملية بالذات بطيئة للغاية. وخلال العمل على إطلاق الإصدار 1.1، بطريقةٍ ما، اكتشفت أن جدول قاعدة البيانات المستخدم من قبل هذه العملية لم تتم فهرسته! ببساطة فهرسة الجدول تحسن الأداء بمقدار 30 مرة لبعض العمليات. لا يعد تعريف فهرس على جدول شائع الاستخدام تحسينا؛ انها مجرد عادة برمجة جيدة.

تكاليف الأداء النسبي للعمليات المشتركة

على الرغم من أنه لا يمكنك الاعتماد على فكرة أن بعض العمليات تكون أكثر تكلفة من غيرها من العمليات من دون قياسها، فإن بعض العمليات تميل إلى أن تكون أكثر تكلفة.

عندما تبحث عن "الدبس" في البرنامج، استخدم الجدول 25-2 للمساعدة في إجراء بعض التخمينات الأولية حول الأجزاء اللزجة لبرنامجك.

الجدول 25-2 كلفة العمليات الشائعة

العملية	مثال	الوقت النسبي المستهلك	
		سي ++	جافا
خط أساس (تخصيص عدد صحيح)	`i = j`	1	1
استدعاءات الإجرائيات			
استدعاء إجرائية دون متغيرات	`foo()`	1	غير متاح
استدعاء إجرائية *خاصة* (private) دون متغيرات	`this.foo()`	1	0.5
استدعاء إجرائية خاصة مع متغير واحد	`this.foo( i )`	1.5	0.5
استدعاء إجرائية خاصة مع متغيرَين	`this.foo( i, j )`	2	0.5
استدعاء إجرائية كائن	`bar.foo()`	2	1
استدعاء إجرائية مشتقة	`derivedBar.foo()`	2	1
استدعاء إجرائية متعددة الأشكال	`abstractBar.foo()`	2.5	2
مراجع الكائنات			
فك ما يشير إليه مرجع بالمستوى 1	`i = obj.num`	1	1
فك ما يشير إليه مرجع بالمستوى 2	`i = obj1.obj2. num`	1	1
كل عملية تفكيك مرجع إضافية	`i = obj1.obj2.obj3...`	غير قابل للقياس	غير قابل للقياس
العمليات على الأعداد الصحيحة			
اسناد عدد صحيح (محلي)	`i = j`	1	1
اسناد عدد صحيح (موروث)	`i = j`	1	1
جمع الاعداد الصحيحة	`i = j + k`	1	1
طرح الاعداد الصحيحة	`i = j - k`	1	1
ضرب الاعداد الصحيحة	`i = j * k`	1	1
قسمة الاعداد الصحيحة	`i = j \ k`	5	1.5
العمليات على الأعداد الحقيقية(الفاصلة العائمة)			
اسناد عدد حقيقي	`x = y`	1	1
جمع الاعداد الحقيقية	`x = y + z`	1	1
طرح الاعداد الحقيقية	`x = y - z`	1	1
ضرب الاعداد الحقيقية	`x = y * z`	1	1
قسمة الاعداد الحقيقية	`x = y / z`	4	1
التوابع الفائقة Transcendental			
الجذر التربيعي لعدد حقيقي	`x = sqrt( y )`	15	4
جيب عدد حقيقي	`x = sin( y )`	25	20
لوغاريتم عدد حقيقي	`x = log( y )`	25	20
e^y كعدد حقيقي	`x = exp( y )`	50	20
المصفوفات			

الدخول إلى مصفوفة أعداد صحيحة عن طريق دليل ثابت	i = a[5]	1	1
الدخول إلى مصفوفة أعداد صحيحة عن طريق دليل متغير	i = a[j]	1	1
الدخول إلى مصفوفة أعداد صحيحة ثناية البعد عن طريق أدلة ثابتة	i = a[3, 5]	1	1
الدخول إلى مصفوفة أعداد صحيحة ثناية البعد عن طريق أدلة متغيرة	i = a[j, k]	1	1
الدخول إلى مصفوفة أعداد حقيقية عن طريق دليل ثابت	x = z[5]	1	1
الدخول إلى مصفوفة أعداد حقيقية عن طريق دليل متغير	x = z[j]	1	1
الدخول إلى مصفوفة أعداد حقيقية ثناية البعد عن طريق أدلة ثابتة	x = z[3, 5]	1	1
الدخول إلى مصفوفة أعداد حقيقية ثناية البعد لعناصر عن طريق أدلة متغيرة	x = z[j, k]	1	1

ملاحظة: القياسات في هذا الجدول حساسة جدا لبيئة الجهاز المحلي، وتحسينات المترجم، والشفرة التي تم إنشاؤها بواسطة مترجمات محددة. القياسات بين سي ++ وجافا غير قابلة للمقارنة مباشرة.

لقد تغيّر الأداء النسبي لهذه العمليات بشكل كبير منذ الإصدار الأول من هذا الكتاب" Code Complete "، لذا إذا كنت تقارب ضبط الشفرة مع أفكار حول الأداء عمرها 10 سنوات، فقد تحتاج إلى تحديث تفكيرك.

معظم العمليات الشائعة تقريباً بنفس الكلفة – استدعاءات الإجرائية، والإسنادات، وحساب الأعداد الصحيحة وحساب الفاصلة العائمة كلها متساوية تقريباً. وظائف الرياضيات الفائقة مكلفة للغاية.

استدعاءات الإجرائيات متعددة الأشكال هي أغلى قليلا من الأنواع الأخرى لاستدعاءات الاجرائيات.

الجدول 25-2، أو ما شابه، هو المفتاح الذي يفتح أقفال جميع تحسينات السرعة الموضحة في الفصل 26. وفي كل الحالات، تأتي السرعة المحسنة من استبدال عملية مكلفة بأخرى أرخص. يقدم الفصل 26 أمثلة على كيفية القيام بذلك.

25. 4 القياس

كون أجزاء صغيرة من برنامج عادة تستهلك حصة غير متناسبة من وقت التشغيل، فقيّم شفرتك للعثور على النقاط الفعالة. بعد العثور على النقاط الفعالة وتحسينها، قم بقياس الشفرة مرة أخرى لتقييم مدى تحسينها.

العديد من جوانب الأداء غير قابلة للحدس. تمثل الحالة السابقة في هذا الفصل، والتي تكون فيها 10 أسطر من الشفرة أسرع وأصغر بشكل ملحوظ من سطر واحد، مثال على أحد الطرق التي يمكن أن تفاجئك الشفرة بها.



لا تساعد الخبرة كثيرًا في التحسين أيضاً. قد تكون خبرة الشخص أتت من آلة أو لغة أو مترجم قديم - وعندما يتغير أي شيء من هذه الأشياء، تتوقف جميع الرهانات. لا يمكنك أبدًا التأكد من تأثير التحسين حتى تقوم بقياس التأثير.

منذ سنوات عديدة قبل الوقت الراهن كتبت برنامجاً يجمع العناصر في مصفوفة. بدت الشفرة الأصلية كالتالي:

مثال سي ++ لشفرة بسيطة تجمع عناصر مصفوفة

```
sum = 0;
for ( row = 0; row < rowCount; row++ ) {
   for ( column = 0; column < columnCount; column++ ) {
      sum = sum + matrix[ row ][ column ];
   }
}
```

كانت هذه الشفرة مستقيمة، ولكن أداء إجرائية الجمع للمصفوفة كان خطيرا، وكنت أعرف أن جميع الدخولات إلى المصفوفة واختبارات الحلقة ستكون تكون باهظة الكلفة. كنت أعرف من دروس علوم الحاسب أنه في كل مرة تدخل فيها الشفرة إلى مصفوفة ثنائية الأبعاد، فإن ذلك يؤدي إلى مضاعفات وإضافات باهظة الكلفة. لمصفوفة 100 × 100، ما مجموعه 10000 من عمليات الضرب والجمع، بالإضافة إلى حمل الحلقة. وبتحويلها إلى تدوين مؤشر، حسبتها، وجدت أنه يمكنني زيادة مؤشر واستبدال عمليات الضرب ال 10000 المكلفة ب 10000 عملية زيادة رخيصة نسبيًا. قمت بتحويل الشفرة إلى تدوين المؤشر بعناية وحصلت على التالي[1]:

مثال سي ++ لمحاولة ضبط شفرة جمع العناصر في مصفوفة

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;
while ( elementPointer < lastElementPointer ) {
sum = sum + *elementPointer++;
}
```

على الرغم من أن الشفرة لم تكن قابلة للقراءة كالشفرة الأولى، خاصة للمبرمجين الذين ليسوا خبراء في سي ++، إلا أنني كنت مسرورًا جدًا بنفسي. بالنسبة لمصفوفة 100×100، حسبت أنني قد وفرت 10000 عملية ضرب والكثير من حمل الحلقات الزائد.

[1] **مزيد من القراءة** ذكر جون بنتلي تجربة مماثلة والذي أدى فيها التحويل إلى مؤشرات إلى الإضرار بالأداء بنسبة 10 بالمئة تقريبًا. كان للتحويل نفسه - في سياق آخر - يحسن في الأداء أكثر من 50 في المئة. انظر "استكشاف البرمجيات: كتابة برامج C كُفأة (بنتلي 1991).

كنت مسرورا جدًا لأنني قررت قياس سرعة التحسين، وهو شيء لم أكن أفعله في ذلك الوقت، وبذلك أتمكن من دعم نفسي بشكل أكبر.

هل تعرف ما وجدته؟[1] لم أجد أي تحسن على الإطلاق. لا مع مصفوفة 100× 100. ولا مع مصفوفة 10 10 x. ولا مع أي حجم للمصفوفة. كنت محبطا كثيرا لأنني تعمقت في شفرة التجميع التي أنشأها المترجم لمعرفة سبب عدم نجاح عملية التحسين الخاصة بي. لدهشتي، اتضح أنني لم أكن أول مبرمج على الإطلاق يحتاج إلى التكرار خلال عناصر مصفوفة - محسّن المترجم قام بالفعل بتحويل مصفوفة الوصول إلى المؤشرات. لقد تعلمت أن النتيجة الوحيدة للتحسين التي يمكنك عادة التأكد منها بدون قياس الأداء هي أنك جعلت من قراءة التعليمات البرمجية أكثر صعوبة. إذا لم تكن الحالة تستدعي دفع كلفة قياس الأداء لتعرف أنها أكثر فعالية، فإنها لا تستدعي دفع كلفة التضحية بالوضوح من أجل الأداء.

القياسات يجب أن تكون دقيقة

تحتاج قياسات الأداء لأن تكون دقيقة[2]. ضبط توقيت برنامجك مع ساعة توقيف stop-watch أو من خلال العد "فيل واحد، فيلان، ثلاثة فيلة" ليس دقيقًا. تعتبر أدوات التوصيف مفيدة، أو يمكنك استخدام ساعة نظامك وإجرائياته التي تسجل الأوقات المنقضية لعمليات الحوسبة.

سواء أكنت تستخدم أداة لشخص آخر أو تكتب شفرتك الخاصة لإجراء القياسات، تأكد من أن تقوم بقياس فقط زمن تنفيذ الشفرة التي تقوم بضبطها. استخدم تكات ساعة وحدة المعالجة المركزية الخاصة ببرنامجك بدلا من توقيت اليوم.

من ناحية أخرى، عندما يقوم النظام بالتبديل من برنامجك إلى برنامج آخر، سيتم فرض ضريبة على أحد إجرائياتك للوقت الذي ينقضي في تنفيذ برنامج آخر. وبالمثل، حاول استيعاب مقدار الحمل الزائد وتكاليف بدء تشغيل البرنامج بحيث لا تخالف لا الشفرة الأصلية ولا محاولات الضبط بشكل غير عادل.

25. 5 التكرار Iteration

[1] لا يتمكن أي مبرمج من التنبؤ أو تحليل مكان وجود اختناقات الأداء بدون بيانات. ليس المهم المكان الذي تعتقد أنها موجودة فيه، فستتفاجأ باكتشافها في مكان آخر. —Joseph M. Newcomer

[2] **إشارة مرجعية** للحصول على مناقشة حول أدوات التوصيف profiling، انظر "ضبط الشفرة" في القسم 30. 3

بمجرد تحديدك لاختناقات الأداء، ستندهش من مدى قدرتك على تحسين الأداء عن طريق ضبط الشفرة. قلما تحصل على تحسين بنسبة 10 أضعاف من تقنية واحدة، وبما أنه يمكنك الجمع بين التقنيات بفعالية؛ استمر في المحاولة، حتى بعد العثور على تقنية شغالة.

ذات مرة قمت بكتابة برمجية لمعيار تشفير البيانات (DES). في الواقع، لم أكتبه مرة واحدة - لقد كتبته حوالي 30 مرة. يتم التشفير وفقًا لـ DES بتشفير البيانات الرقمية بحيث لا يمكن فك تشفيرها بدون كلمة مرور. كانت خوارزمية التشفير معقدة جدًا بحيث تبدو وكأنها تم استخدامها على نفسها. كان هدف الأداء لتطبيق DES هو تشفير ملف K18 في 37 ثانية على حاسب IBM الأصلي. نفذ تطبيقي الأول في 21 دقيقة و40 ثانية، لذلك كان لدي طريق عمل طويل.

على الرغم من أن معظم التحسينات الفردية كانت صغيرة، إلا أن التحسينات التراكمية كانت كبيرة. للحكم من خلال النسبة المئوية للتحسينات، أجد أنه لم تكن ثلاثة تحسينات أو حتى أربعة تحقق هدف الأداء المنشود. لكن المجموع النهائي للتحسينات كان فعالا. العبرة من القصة هي أنه إذا قمت بالتنقيب عميقا بما فيه الكفاية، فستتمكن من تحقيق بعض المكاسب المفاجئة.

إن عملية ضبط الشفرة الذي قمت بها في هذه الحالة هي من أكثر حالات ضبط الشفرة صعوبة التي قمت بها على الإطلاق. في نفس الوقت، كانت الشفرة النهائية من أقل الشفرات قابليةً للقراءة وللصيانة من بين الشفرات التي كتبتها على الإطلاق. كانت الخوارزمية الأولية معقدة. كانت الشفرة الناتجة عن تحول اللغة عالية المستوى بالكاد قابلة للقراءة. أنتجت الترجمة إلى المجمِّع إجرائية واحدة مؤلفة من 500 سطر والتي كنت أخاف أن أنظر إليها. بشكل عام، تظل هذه العلاقة بين ضبط الشفرة البرمجية وجودة الشفرة صحيحة. في ما يلي جدول يعرض سجل بالأداء[1]:

التحسين Improvement	علامة الوقت	Optimization الموائمة
-	21:40	التحقيق في البداية– مباشر
٪65	7:30	تحويل من حقول البتات إلى مصفوفات
٪20	6:00	نشر أعمق حلقة for
٪10	5:24	إزالة *التباديل* النهائية
٪ 5	5:06	الجمع بين اثنين من المتغيرات
٪12	4:30	استخدم هوية منطقية للجمع بين أول خطوتين من خوارزمية معيار تشفير البيانات

[1] **إشارة مرجعية** التقنيات المذكورة في هذا الجدول هي الموصوفة في الفصل 26، "تقنيات ضبط الشفرة".

مشاركة متغيرين لنفس الذاكرة لتقليل تناقل البيانات في حلقة داخلية	3:36	٪20
مشاركة متغيرين لنفس الذاكرة لتقليل تناقل البيانات في حلقة خارجية	3:09	٪13
فك Unfold كل الحلقات واستخدام تعليمات مصفوفة محرفية	1:36	٪49
إزالة استدعاءات الإجرائية ووضع كامل الشفرة "في السطر"	0:45	٪53
إعادة كتابة كامل الإجرائية في المجمع	0:22	٪51
النهائي	0:22	٪98

ملاحظة: لا يعني التقدم المنتظم في عمليات التحسين في هذا الجدول أن جميع عمليات التحسين تعمل. لم أظهر كل الأشياء التي جربتها والتي أدت إلى مضاعفة وقت التشغيل. فما لا يقل عن ثلثي التحسينات التي جربتها لم تنجح.

25. 6 ملخص نهج ضبط الشفرة

يجب عليك اتخاذ الخطوات التالية أثناء التفكير فيما إن كان ضبط الشفرة يمكن أن يساعدك في تحسين أداء برنامج:

1. طور البرنامج باستخدام شفرة جيدة التصميم سهلة الفهم والتعديل.
2. في حال كان الأداء ضعيفًا،

ا. احفظ نسخة شغالة من الشفرة بحيث يمكنك من خلالها استرجاع "آخر حالة جيدة معروفة".

ب. قيّم النظام للعثور على النقاط الفعالة.

ج. حدد إن كان سبب الأداء الضعيف يأتي من عدم كفاية التصميم أو أنواع البيانات أو الخوارزميات وإن كان ضبط الشفرة البرمجية مناسبًا أم لا. إذا لم يكن توليف الشفرة مناسبًا، فعد إلى الخطوة 1.

د. حسّن الاختناق المحدد في الخطوة (ج).

هـ. قس كل تحسين واحد في كل مرة.

ي. إذا لم يؤد التحسين إلى تحسين الشفرة، فاسترجع الشفرة التي قمت بحفظها في الخطوة (أ). (عادة، أكثر من نصف محاولات التوليف ستؤدي فقط إلى تحسن لا يذكر في الأداء أو إلى انخفاض الأداء.)

3. كرر الخطوات من الخطوة 2.

مصادر إضافية[1]

يحتوي هذا القسم على الموارد المتعلقة بتحسين الأداء بشكل عام. للحصول على موارد إضافية والتي تناقش تقنيات محددة لضبط الشفرة، راجع قسم "الموارد الإضافية" في نهاية الفصل 26.

[1] cc2e.com/2585

الأداء

Smith, Connie U. and Lloyd G. Williams. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Boston, MA: Addison-Wesley, 2002.

"حلول الأداء": دليل عملي لإنشاء برامج سريعة الاستجابة وقابلة للتطوير:

يغطي هذا الكتاب هندسة أداء البرمجيات، ونهج لبناء الأداء في أنظمة البرامج في جميع مراحل التطوير. فهو يستخدم نطاق واسع من الأمثلة وحالات دراسة لأنواع متعددة من البرامج. ويتضمن توصيات محددة لتطبيقات الويب ويولي اهتماما خاصا لقابلية التطوير.

Newcomer, Joseph M. “Optimization: Your Worst Enemy.”[1] May 2000, www.flounder.com/optimization.htm"التحسين: أسوء أعداءك "

نيوكامر هو مبرمج نظم خبير يصف المخاطر المختلفة لاستراتيجيات التحسين غير الفعالة بالتفاصيل البيانية.

الخوارزميات وأنواع البيانات[2]

Knuth, Donald. *The Art of Computer Programming"فن برمجة الحاسب"*, vol. 1, Fundamental Algorithms"الخوارزميات الأساسية", 3d ed. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald. The Art of Computer Programming" فن برمجة الحاسب ", vol. 2, Seminumerical Algorithms" الخوارزميات العددية الجزئية" 3d ed. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald. The Art of Computer Programming" فن برمجة الحاسب ", vol. 3, Sorting and Searching"الفرز والبحث", 2d ed. Reading, MA: Addison-Wesley, 1998.

كانت هي المجلدات الثلاثة الأولى من سلسلة كان من المفترض أن تصل إلى سبعة مجلدات. يمكن أن تكون مرعبة بعض الشيء. بالإضافة إلى وصف الخوارزميات باللغة الإنجليزية، تم وصفها في التدوين الرياضي أو MIX، وهي لغة التجميع للحاسب الخيالي MIX. تحتوي هذه الكتب على تفاصيل شاملة حول عدد كبير من الموضوعات، وإذا كان لديك اهتمام كبير بخوارزمية معينة، فلن تجد مرجعًا أفضل.

Sedgewick, Robert. *Algorithms in Java"الخوارزميات في جافا", Parts 1-4*, 3d ed. Boston, MA: Addison-Wesley, 2002.

[1] cc2e.com/2592

[2] cc2e.com/2599

تحتوي الأجزاء الأربعة من الكتاب على استبيان لأفضل النُهج لحل مجموعة واسعة من المشاكل. وتشمل مجالات موضوعه: الأساسيات، والفرز، والبحث، وتنفيذ نوع البيانات المجردة، ومواضيع متقدمة.

Sedgewick's Algorithms in Java"",الخوارزميات في جافا Part 5, 3d ed. (2003)

تغطي خوارزميات الرسم البياني

Sedgewick's Algorithms in C++"الخوارزميات في سي++",Parts 1-4, 3d ed. (1998), Algorithms in C++"الخوارزميات في سي ++", Part 5, 3d ed.

(2002), Algorithms in C"الخوارزميات في سي", Parts 1-4, 3d ed. (1997), and Algorithms in C, Part 5, 3d ed. (2001)

نظمت أيضا بطريقة مشابهة. كان سيدجويك طالب بروفيسوراه عند نَث.

قائمة التحقق: استراتيجيات ضبط شفرة

أداء البرنامج الكلي

- هل فكرت في تحسين الأداء من خلال تغيير متطلبات البرنامج؟
- هل فكرت في تحسين الأداء عن طريق تعديل تصميم البرنامج؟
- هل فكرت في تحسين الأداء عن طريق تعديل تصميم الصفوف؟
- هل فكرت في تحسين الأداء عن طريق تجنب التفاعلات مع نظام التشغيل؟
- هل فكرت في تحسين الأداء عن طريق تجنب الإدخال / الإخراج؟
- هل فكرت في تحسين الأداء باستخدام لغة مترجمة بدلاً من لغة مفسرة؟
- هل فكرت في تحسين الأداء باستخدام تحسينات المترجم؟
- هل فكرت في تحسين الأداء من خلال التبديل إلى أجهزة أخرى مختلفة؟
- هل فكرت في ضبط الشفرة فقط كملاذ أخير؟

نهج ضبط الشفرة

- هل برنامجك صحيح تمامًا قبل أن تبدأ بضبط الشفرة؟
- هل قمت بقياس اختناقات الأداء قبل البدء بضبط الشفرة؟
- هل قمت بقياس تأثير كل تغيير في ضبط الشفرة؟
- هل قمت بالتراجع عن تغييرات ضبط الشفرة التي لم تنتج التحسين المطلوب؟
- هل جربت أكثر من تغيير لتحسين أداء كل اختناق - أي، هل كررت؟

نقاط مفتاحية

- إن الأداء هو جانب واحد فقط من جوانب الجودة العامة للبرمجية، وهو عادة ليس الجانب الأكثر أهمية. تمثل الشفرة التي تم ضبطها بدقة جانبًا واحدًا فقط للأداء الكلي، وهي عادةً ليست الأكثر أهمية. عادةً ما يكون لهيكلة البرنامج والتصميم التفصيلي وبنية البيانات واختيار الخوارزمية تأثيرًا أكبر على سرعة وحجم تنفيذ البرنامج، مقارنةً بفعالية شفرته.
- القياس الكمي هو مفتاح لزيادة الأداء. وهو لازم لتحديد المناطق التي سيتم فيها تحسين الأداء بشكل فعلي، ولازم أيضا إلى التحقق من أن التحسينات تعمل على تحسين البرمجية بدلاً من إضعافها.
- معظم البرامج تستهلك معظم وقتها في جزء صغير من شفرتها. لن تعرف الشفرة المطلوبة حتى تقوم بتقييمها.
- عادة ما تكون هناك حاجة إلى تكرارات متعددة لتحقيق تحسينات الأداء المطلوبة من خلال ضبط الشفرة.
- الطريقة الأفضل للتحضير لأعمال الأداء أثناء كتابة الشفرة الأولية هي كتابة شفرة نظيفة يسهل فهمها وتعديلها.

القسم السادس: اعتبارات النظام

في هذا القسم:

الفصل 27

كيف يُؤثر حجم البرنامج على عملية البناء

المحتويات [1]

مواضيع ذات صلة

- المتطلبات الأساسية لعملية البناء: الفصل 3
- تحديد نوع البرمجية التي تعمل عليها: القسم 2.3
- إدارة البناء: الفصل 28

لا يقتصر التوسّع في تطوير البرمجيات على مجرد تنفيذ مشروع صغير وجعل كل جزء منه أكبر. افترض أنك تكتب حزمةً برمجيةً "جيجاترون" (Gigatron) ذات 25000 سطر في 20 شهر عمل، ووجدت 500 خطأ في اختبار الحقل. افترض أن جيجاترون 1.0 يعمل بنجاح، كما في جيجاترون 2.0، وبدأت العمل على جيجاترون ديلوكس (Gigatron Deluxe)، فمن المتوقع أن تتكون هذه النسخة المُحسّنة من البرنامج مكونة من 250000 سطر برمجي.

على الرغم من أن حجم النسخة المحسنة من جيجاترون أكبر من النسخة الأصلية بعشرة أضعاف، إلا أن جيجاترون ديلوكس لن يأخذ 10 أضعاف الجهد المطلوب لتطويره؛ بل سيأخذ 30 ضعف من الجهد. علاوة على ذلك، فإن 30 ضعف من مجمل الجهد لا ينطوي على 30 ضعف مما تتطلبه عملية البناء. ربما ينطوي على 25

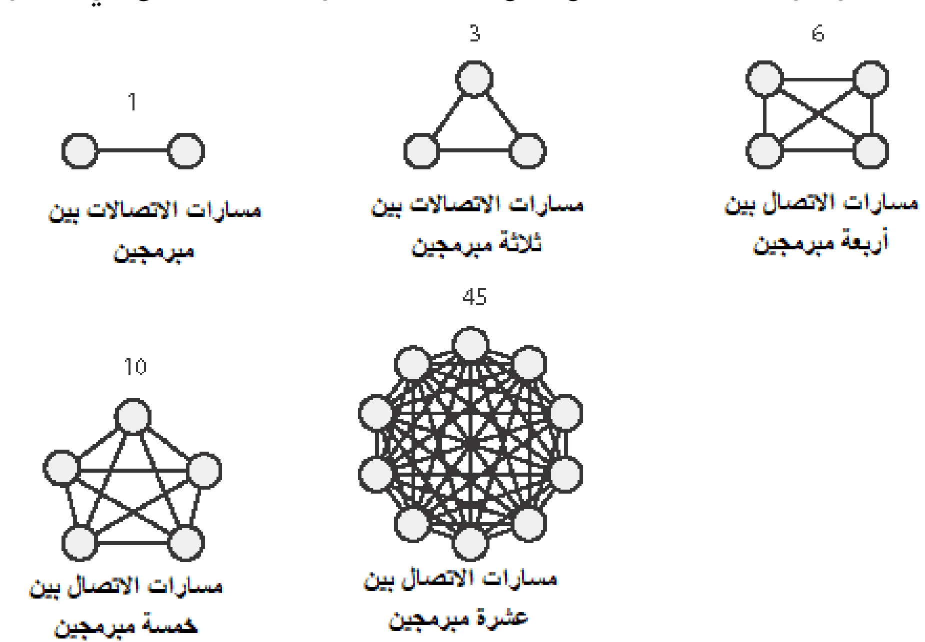
[1] cc2e.com/2761

ضعف من عملية البناء و40 ضعف من الهيكلة واختبار النظام. ولن يكون لديك 10 أضعاف من الأخطاء أيضًا؛ سيكون لديك 15 ضعفًا – أو أكثر.

إذا كنت معتادًا على العمل في المشروعات الصغيرة، فيمكن أن يخرج مشروعك الأول المتوسط -الكبير إلى خارج نطاق السيطرة، ليصبح وحشًا لا يمكن السيطرة عليه، بدلًا من النجاح الرائع الذي كنت تتخيله. يخبرك هذا الفصل عن نوع الوحش المُتوقع وعن مكان إيجاد السوط والكرسي لترويضه. في المقابل، إذا كنت معتادًا على العمل في مشاريع كبيرة، فقد تستخدم أساليبًا غير لازمة في مشروع صغير. يصف هذا الفصل كيف يمكنك الاقتصاد للحفاظ على مشروع صغير من التداعي للسقوط تحت تأثير وزن النفقات الخاصة به.

27. 1 الاتصالات والحجم

إذا كنت الشخص الوحيد في المشروع، فإن مسار الاتصالات الوحيد هو بينك وبين العميل، ما لم تحسب المسار عبر الجسم الثفني، وهو المسار الذي يربط الجانب الأيسر من دماغك بالأيمن. ولكن مع زيادة عدد الأشخاص في المشروع[1]، يزداد أيضًا عدد مسارات الاتصالات. ولكن لا يزداد الرقم بشكل تصاعدي مع زيادة عدد الأشخاص. بل يزداد بالتناسب مع مربع عدد الأشخاص، كما هو موضح في الشكل 27-1.



الشكل 27-1 يزداد عدد مسارات الاتصالات بشكل تصاعدي بما يتناسب مع مربع عدد الأشخاص في الفريق.



كما ترى، فإن مشروع مكون من شخصين يمتلك مسار اتصال واحد فقط. ولدى مشروع مكون من خمسة أشخاص 10 مسارات. ويحتوي المشروع المُكوّن من عشرة أشخاص على 45 مسارًا، على

[1] **هامش**: الجسم الثفني: هو حزمة مسطحة واسعة من الألياف العصبية تحت القشرة في الشق الطولي للدماغ، ويتكون من ملايين الألياف العصبية التي تربط بين النصفين الكرويين الأيمن والأيسر في الدماغ ويسهل الاتصال بينهما وتنتقل المعلومات بينهما على هيئة إشارات كهربائية.

افتراض أن كل شخص يتكلم مع كل الأشخاص الآخرين. لدى عشرة في المئة من المشاريع التي لديها 50 أو أكثر من المبرمجين ما لا يقل عن 1200 من المسارات المحتملة. ومع زيادة مسارات الاتصال لديك، يصرف المزيد من الوقت الذي تقضيه في التواصل وبالتالي المزيد من الفرص لأخطاء التواصل. حيث تتطلب المشاريع كبيرة الحجم تقنيات تنظيمية تعمل على تبسيط الاتصال أو الحد منه بطريقة معقولة. يتمثل النهج النموذجي المُتبع لتبسيط الاتصال في إضفاء الطابع الرسمي عليه في الوثائق.

فبدلًا من وجود 50 شخصًا يتحدثون مع بعضهم البعض في كل تجمّع يمكن تصوره، فإن هؤلاء الـ 50 شخصًا يقرؤون ويكتبون المستندات. بعضها عبارة عن مستندات نصية؛ وبعضها عبارة عن رسومات. وبعضها مطبوع على الورق؛ ويحتفظ ببعضها الآخر بشكل إلكتروني.

2 .27 نطاق أحجام المشروع

هل حجم المشروع الذي تعمل عليه هو حجم نموذجي؟ النطاق الواسع لأحجام المشاريع يعني أنه لا يمكنك التفكير في أي حجم منفرد ليكون نموذجيًا. إحدى طرق التفكير في حجم المشروع هي التفكير في حجم فريق المشروع. فيما يلي تقدير تقريبي للنسب المئوية لجميع المشروعات التي يتم تنفيذها بواسطة فرق ذات أحجام مختلفة:

حجم الفريق	النسبة التقريبية للمشروعات
3–1	25%
10–4	30%
25–11	20%
50–26	15%
+50	10%

المصدر: مقتبس من "استقصاء لممارسة هندسة البرمجيات: الأدوات والطرق والنتائج" (1983 بيك وبيركنز)، *النظم الإيكولوجية لتطوير البرمجيات الذكية* (هايسميث 2002)، *وموازنة المرونة والانضباط* (بويم وتيرنر 2003).

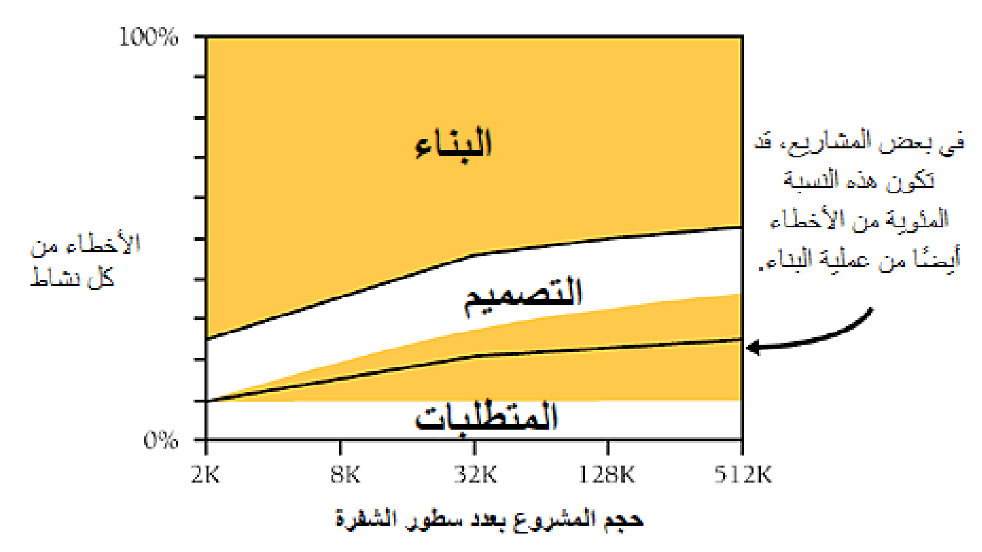
أحد جوانب البيانات حول حجم المشروع التي قد لا تظهر على الفور هو الفرق بين النسب المئوية للمشروعات ذات الأحجام المختلفة وعدد المبرمجين الذين يعملون في مشاريع من كل حجم. ولأن المشاريع الكبيرة تستخدم عدد أكبر من المبرمجين في كل مشروع أكثر مما تستخدمه المشروعات الصغيرة، فهي توظف نسبة كبيرة من جميع المبرمجين. فيما يلي تقدير تقريبي لنسبة كل المبرمجين الذين يعملون في مشاريع بأحجام مختلفة:

حجم الفريق	النسبة التقريبية للمبرمجين
3–1	5%
10–4	10%
25–11	15%
50–26	20%
+50	50%

المصدر: مقتبس من "استقصاء لممارسة هندسة البرمجيات: الأدوات والطرق والنتائج" (1983 بيك وبيركنز)، *النظم الإيكولوجية لتطوير البرمجيات الذكية* (هايسميث 2002)، و*موازنة الرشاقة والانضباط* (بويم وتيرنز 2003).

27. 3 تأثير حجم مشروع على الأخطاء

تتأثر كل من كمية ونوع الأخطاء بحجم المشروع[1]. قد لا تعتقد أن نوع الخطأ سيتأثر ولكن مع زيادة حجم المشروع، عادةً ما يعزى نسبة أكبر من الأخطاء إلى أخطاء في المتطلبات والتصميم، كما هو موضح بالشكل 2-27.



الشكل 2-27 مع زيادة حجم المشروع، تأتي الأخطاء عادةً من المتطلبات والتصميم. وفي بعض الأحيان، لا تزال الأخطاء تأتي في المقام الأول من عملية البناء (بويم 1981، جرادي 1987، جونز 1998).

بيانات مثبتة

في المشاريع الصغيرة، تُشكل أخطاء البناء حوالي 75 في المئة من مجموع الأخطاء التي يُعثر عليها. لدى هذه المنهجية تأثير أقل على جودة الشفرة، وكثيرًا ما يكون التأثير الأكبر على جودة البرنامج هو المهارة الفردية في كتابة البرنامج (جونز 1998).

في المشاريع الكبيرة، يمكن أن تتناقص أخطاء البناء تدريجيا إلى حوالي 50 في المئة من مجموع الأخطاء؛ حيث تُشكل أخطاء المتطلبات والهيكلة الفرق. ومن المحتمل أن هذا يرتبط بحقيقة أن تطوير المتطلبات وتصميم الهيكلة مطلوبان أكثر في المشاريع الكبيرة، وبالتالي فإن فرصة حدوث أخطاء ناجمة عن هذه الأنشطة أكبر نسبيًا. ومع ذلك، في بعض المشاريع الكبيرة جدًا، فإن نسبة أخطاء البناء لا تزال مرتفعة؛ في بعض الأحيان

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن الأخطاء، انظر القسم 22. 4 "الأخطاء النموذجية"

حتى مع 500000 سطر من الشفرة، يمكن أن يعزى ما يصل إلى 75 في المئة من الأخطاء إلى عملية البناء (جرادي 1987).



وكما تتغير أنواع العيوب مع تغير الحجم، فإن أعداد العيوب تتغير أيضًا. من الطبيعي أن تتوقع لمشروع ذو حجم ضعف مشروع آخر أن يكون لديه ضعف عدد الأخطاء. ولكن كثافة العيوب – عدد العيوب لكل 1000 سطر من الشفرة – تزداد. فمن المرجّح أن يواجه المنتج مضاعف الحجم أكثر من ضعف عدد الأخطاء. يوضح الجدول 27-1 مدى كثافة العيوب التي يمكنك توقعها في المشروعات ذات الأحجام المختلفة.

جدول 27-1 حجم المشروع وكثافة الخطأ النموذجية[1]	
حجم المشروع (عدد الأسطر في الشفرة)	كثافة الخطأ النموذجية
أقل من K2	0–25 خطأ في كل ألف سطر من الشفرة (KLOC)[2]
2K – 16K	0–40 خطأ في كل ألف سطر من الشفرة
16K– 64K	0.5–50 خطأ في كل ألف سطر من الشفرة
64K – 512K	2–70 خطأ في كل ألف سطر من الشفرة
512K أو أكثر	4–100 خطأ في كل ألف سطر من الشفرة
المصادر: "جودة البرامج وإنتاجية المبرمج" (جونز 1977)، تقدير تكاليف البرمجيات (جونز 1998).	

أخذت البيانات الواردة في هذا الجدول من مشاريع محددة، وقد لا تحمل هذه الأرقام إلا القليل من الشبه لتلك الخاصة بالمشاريع التي عملتَ عليها. لكن هذه البيانات موضحة كصورة سريعة لهذه الصناعة. وهذا يشير إلى أن عدد الأخطاء يزداد بشكل كبير مع زيادة حجم المشروع، حيث مع المشاريع الكبيرة جدًا يصل عدد الأخطاء في كل ألف سطر من أسطر الشفرة إلى أربعة أضعاف مقارنة بالمشاريع الصغيرة. وسيحتاج مشروع كبير للعمل بصعوبة أكثر من مشروع صغير لتحقيق معدل الخطأ نفسه

4.27 تأثير حجم مشروع على الإنتاجية

تملك الإنتاجية الكثير من القواسم المشتركة مع جودة البرامج عندما يتعلق الأمر بحجم المشروع. ففي المشاريع ذات الأحجام الصغيرة (2000 سطر من الشفرة أو أصغر)، فإن أكبر تأثير على الإنتاجية هو مهارة المبرمج الفردية (جونز 1998). ومع زيادة حجم المشروع، يصبح تأثير حجم الفريق وتنظيمه أكبر على الإنتاجية.

---

[1] . **إشارة مرجعية**: تُمثل البيانات في هذا الجدول متوسط الأداء. من المفيد للمنظمات أن تملك معدلات خطأ أفضل من الحد الأدنى المبين هنا. على سبيل المثال، انظر "ما هو عدد الأخطاء التي من الممكن توقع إيجادها" في القسم 4.22.

[2] **هامش**: KLOC (kilo of lines of code) ألف سطر من الشفرة

بيانات مثبتة

ما هو حجم المشروع المطلوب قبل أن يبدأ حجم الفريق في التأثير على الإنتاجية؟ في " النماذج الأولية مقابل توظيف خبرة مشاريع متعددة"، أفاد بويم وغراي وسيوالدت بأن الفِرق الصغيرة أنجزت مشاريعها مع زيادة إنتاجيتها بنسبة 39٪ مقارنة بالفرق الأكبر. وما هو الحجم المطلوب للفريق؟ شخصين للمشروعات الصغيرة وثلاثة للمشاريع الكبيرة (1984). يعطي الجدول 27-2 السبق الصحفي للعلاقة العامة بين حجم المشروع والإنتاجية.

الجدول 27-2 حجم المشروع والإنتاجية	
حجم المشروع (عدد الأسطر في الشفرة)	عدد أسطر الشفرة لكل سنة عامل (القيمة الاسمية كوكومو Cocomo II[1] في الأقواس)
K1	2,500–25,000 (4,000)
K10	2,000–25,000 (3,200)
K100	1,000–20,000 (2,600)
K1000	700–10,000 (2,000)
K10000	300–5,000 (1,600)
المصدر: مشتقة من البيانات في مقاييس التميّز (بوتنام ومايرز 1992)، برامج القوة الصناعية (بوتنام ومايرز 1997)، تقدير تكلفة البرمجيات مع نموذج تكلفة البناء Cocomo II (بويم وآخرون 2000)، و" تطوير البرمجيات في جميع أنحاء العالم: حالة الممارسة" (كوسومانو وآخرون 2003).	

تتحدد الإنتاجية فعليًا من خلال نوع البرامج التي تعمل عليها، وجودة الموظفين، ولغة البرمجة، والمنهجية، وتعقيد المُنتج، وبيئة البرمجة، ودعم الأداة، وكيفية حساب "عدد أسطر الشفرة"، وكيفية تضمين جهد الدعم من قبل غير المبرمجين في " عدد أسطر الشفرة لكل “سنة عامل” والعديد من العوامل الأخرى، لذلك تختلف الأرقام المحددة في الجدول 27-2 بشكل كبير.

ومع ذلك، اِفهم أن الاتجاه العام الذي تظهره الأرقام مهم. حيث يمكن أن تبلغ الإنتاجية في المشاريع الصغيرة من 2-3 ضعف أكثر من الإنتاجية في المشروعات الكبيرة، ويمكن أن تختلف الإنتاجية بعامل من 5-10 من أصغر المشاريع إلى أكبرها.

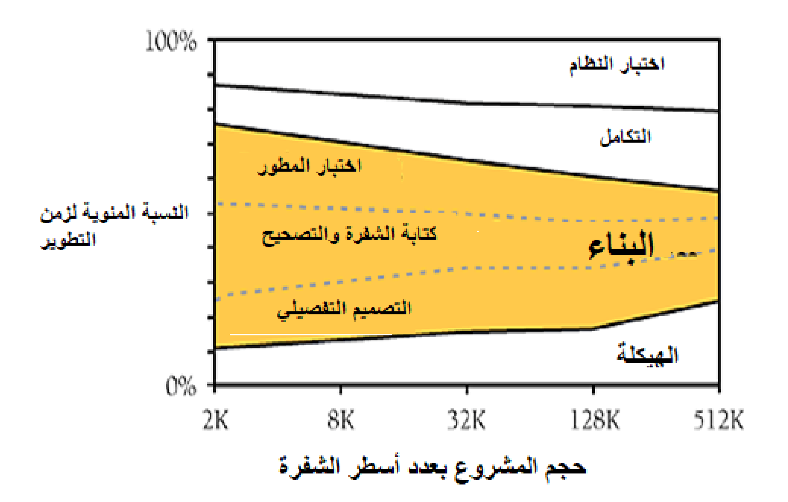
27. 5 تأثير حجم مشروع على أنشطة التطوير

[1] هامش: كوكومو اختصار لكلمة COnstructive COst MOdel. وهي إحدى طرق هندسة البرمجيات، وتستعمل لتقدير الجهد اللازم لإنتاج برنامج معين. وهو قانون امبيري تم استخلاصه من ملاحظة الجهد اللازم لصناعة البرامج.

إذا كنت تعمل في مشروع من شخص واحد، فإن المؤثر الأكبر على نجاح المشروع أو فشله هو أنت. أما إذا كنت تعمل على مشروع يتألف من 25 شخصًا، فمن الممكن أن تظل الأكثر تأثيرا، ولكن من المرجح ألا يرتدي أي شخص وسام الشرف لهذا التميّز؛ بل سيكون لمؤسستك تأثير أقوى على نجاح أو فشل المشروع.

نسب النشاط والحجم

مع زيادة حجم المشروع والحاجة إلى الاتصالات الرسمية، فإن أنواع الأنشطة التي يحتاجها مشروع تتغير بشكل كبير. يوضح الشكل 27-3 نسب أنشطة التطوير للمشاريع ذات الأحجام المختلفة.

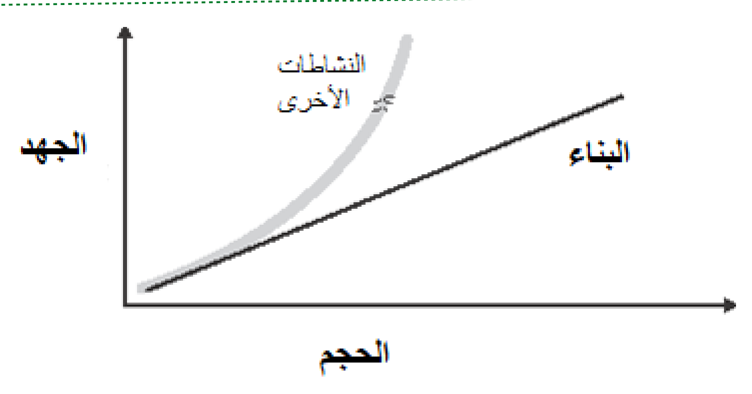


الشكل 27-3: تهيمن أنشطة البناء على المشاريع الصغيرة. تتطلب المشاريع الكبيرة مزيدًا من الهيكلة، وأعمال التكامل، واختبار النظام لتحقيق النجاح. لا يتم عرض متطلبات العمل على هذا الرسم البياني لأن جهد المتطلبات لا يكون تابع بشكل مباشر لحجم البرنامج كما هي الأنشطة الأخرى (ألبرشت 1979؛ كلاس 1982؛ بويم، كراي وسيوالدت 1984؛ بودي 1987؛ كارد 1987؛ ماكغري، واليغورا، وماكديرموت 1989؛ بروكس 1995 ؛ Jones 1998 ؛ جونز 2000 ؛ بويم وآخرون 2000).



في مشروع صغير، يُعد البناء النشاط الأبرز حتى الآن، حيث يستهلك ما يصل إلى 65٪ من إجمالي وقت التطوير. وفي مشروع متوسط الحجم، لا يزال البناء هو النشاط المهيمن، لكن حصته من إجمالي الجهود تنخفض إلى حوالي 50٪. في المشروعات الضخمة، تستغرق الهيكلة، والتكامل، واختبار النظام المزيد من الوقت ويصبح البناء أقل هيمنة. باختصار، مع زيادة حجم المشروع، يُصبح البناء جزءًا أصغر من مجمل الجهد. يبدو الرسم البياني كما لو كنت تستطيع تمديده إلى اليمين وجعل عملية البناء تختفي تمامًا، لذا وحرصًا على حماية وظيفتي، أوقفتها عند حجم 512 كيلوبايت.

تصبح عملية البناء أقل سيطرةً لأنه مع زيادة حجم المشروع، فإن أنشطة البناء - التصميم التفصيلي وكتابة الشفرة وتصحيح الأخطاء واختبار الوحدة - تزداد تصاعديا بشكل متناسب ولكن العديد من الأنشطة الأخرى تزداد بشكل أسرع. يقدم الشكل 27-4 مثالًا توضيحيًا.



الشكل 4-27 تعتبر كمية أعمال بناء البرمجيات تابع شبه خطي لحجم المشروع. أما الأنواع الأخرى من العمل تزداد بشكل غير خطي مع زيادة حجم المشروع.

ستقوم المشاريع القريبة في الحجم بأنشطة مماثلة، ولكن عند اختلاف أحجامها، سوف تختلف أنواع الأنشطة أيضًا. كما وصفت مقدمة هذا الفصل، عندما يتم إصدار جيغاترون ديلوكس بمقدار 10 أضعاف حجم جيغاترون الأصلي، فإنه سيكون هناك حاجة إلى 25 ضعف من الجهد الإضافي لعملية البناء، و25-50 ضعف من جهد التخطيط، و30 ضعف من جهد عملية التكامل، و40 ضعف من عملية الهيكلة واختبار النظام.

بيانات مثبتة

تختلف نسبة الأنشطة لأن أنشطة مختلفة تصبح حاسمة في حجوم مختلفة للمشاريع. وجد باري بويم وريتشارد تيرنر أن إنفاق حوالي خمسة بالمائة من إجمالي تكاليف المشروع على الهيكلة والمتطلبات أنتج أقل تكلفة للمشاريع ذات حجم في نطاق 10000 سطر من الشفرة. ولكن بالنسبة للمشاريع في نطاق 100.000 سطر من الشفرة، فإن إنفاق 15-20٪ من جهد المشروع على الهيكلة والمتطلبات قد حقق أفضل النتائج (بويم وتيرنر 2004).

فيما يلي قائمة الأنشطة التي تنمو بمعدل أكثر من التناسب الخطي مع زيادة حجم المشروع:

- الاتصالات
- التخطيط
- الإدارة
- تطوير المتطلبات
- التصميم الوظيفي للنظام
- تصميم الواجهة والمواصفات
- الهيكلة
- التكامل

- إزالة العيوب
- اختبار النظام
- إنتاج الوثائق

وبغض النظر عن حجم المشروع، فإن بعض التقنيات تكون دومًا قيّمة: أنشطة التشفير المُنضبط، وتصميم وفحص الشفرة من قبل مطورين آخرين، وأدوات الدعم الجيدة، واستخدام لغات عالية المستوى. هذه التقنيات قيّمة في المشاريع الصغيرة ولا تقدر بثمن في المشاريع الكبيرة.

البرامج والمنتجات والأنظمة ومنتجات النظام

لا يمثل عدد أسطر الشفرة وحجم الفريق العوامل المؤثرة الوحيدة على حجم المشروع[1]. حيث أن التأثير الأكثر دقةً هو جودة وتعقيد البرامج النهائية. ربما استغرق جيغاترون الأصلي، جيغاترون جي أر Gigatron Jr شهرًا واحدًا فقط للكتابة والتصحيح. كان برنامجًا واحدًا كُتبَ واختُبر ووُثّق من قبل شخص واحد. إذا كان جيغاترون جي أر البالغ طوله 2500 سطر قد استغرق شهرًا واحدًا من قبل شخص واحد، فلماذا استغرق جيغاترون البالغ طوله 25 ألف سطر 20 شهرًا؟ أبسط نوع من البرمجيات هو "برنامج" واحد يستخدمه نفس الشخص الذي قام بتطويره، أو يُستخدم بشكل غير رسمي من قِبل البعض.

أما النوع الأكثر تعقيدًا من البرامج هو برنامج "منتج"، وهو برنامج مخصص للاستخدام من قِبل أشخاص آخرين غير المطوّر الأصلي. يتم استخدام البرنامج "المنتج" في بيئات تختلف عن البيئة التي تم إنشاء المنتج فيها. حيث يتم اختباره على نطاق واسع قبل إصداره، ويتم توثيقه، وهو قابل للصيانة من قبل الآخرين. حيث يُكلف البرنامج "المُنتج" حوالي ثلاثة أضعاف تطويره كـ "برنامج حاسوبي".

ويُطلب مستوى آخر من الخبرة لتطوير مجموعة من البرامج التي تعمل معًا. يُطلق على هذه المجموعة اسم "نظام". إن تطوير "نظام" أكثر تعقيدًا من تطوير برنامج بسيط بسبب تعقيد تطوير الواجهات بين الأجزاء المنفصلة والعناية اللازمة لدمج هذه الأجزاء. على العموم، يُكلف "النظام" حوالي ثلاثة أضعاف تكلفة البرنامج البسيط.

بيانات مثبتة

عندما يتم تطوير "منتج النظام"، فإنه يحتوي على صقل منتج وأجزاء متعددة من النظام. تكلفة منتجات النظام حوالي تسعة أضعاف البرامج البسيطة (بروكس 1995، شول وآخرون.2002).

إن الفشل في تقدير الاختلافات في الصقل والتعقيد بين البرامج، والمنتجات، والأنظمة، ومنتجات النظام هو حالة شائعة لأخطاء التقدير. قد يقدر المبرمجون الذين يستخدمون خبرتهم في بناء برنامج الجدول الزمني

[1] **قراءة متعمقة**: للحصول على تفسير آخر لهذه النقطة، انظر الفصل الأول في رجل الشهر الأسطوري (The Mythical Man-Month) (بروكس 1995).

لبناء منتج نظام بأقل من قيمته بما يقرب من معامل 10. عندما تدرس المثال التالي، ارجع إلى الرسم البياني في الشكل 27-3. إذا استخدمت خبرتك في كتابة شفرة بحجم 2 كيلو سطر لتقدير الوقت الذي ستحتاجه لتطوير برنامج بحجم 2 كيلو، فإن تقديرك سيمثل 65 بالمائة فقط من إجمالي الوقت الذي ستحتاج إليه فعليًا لتنفيذ جميع الأنشطة التي تدخل في تطوير البرنامج. لا تستغرق كتابة شفرة من 2000 سطر وقت إنشاء برنامج كامل يحتوي 2000 سطر من الشفرة. إذا لم تفكر في الوقت الذي تستغرقه للقيام بأنشطة غير البناء، فسوف يستغرق التطوير 50٪ من الوقت أكثر مما تتوقعه.

كلما تقدمت، يصبح البناء جزءًا أصغر من مجمل الجهد المبذول في المشروع. إذا كنت تستند بتقديرك فقط على تجربة البناء، فسوف يزيد خطأ التقدير. إذا كنت قد استخدمت خبرة البناء ذات ال2000 سطر برمجي لتقدير الوقت المستغرق لتطوير برنامج 32 كيلو بايت، فإن تقديرك سيكون 50٪ فقط من الوقت الإجمالي المطلوب؛ وسوف يستغرق التطوير أكثر 100٪ من الوقت مما تتوقعه.

سيكون خطأ التقدير هنا منسوبًا تمامًا إلى عدم فهمك لتأثير الحجم على تطوير البرامج الأكبر. إذا فشلت أيضًا في التفكير في الدرجة الإضافية من الصقل المطلوب لمنتج بدًلا من مجرد برنامج، فيمكن أن يزيد الخطأ بسهولة بعامل ثلاثة أو أكثر.

المنهجية والحجم

يتم استخدام المنهجيات في المشاريع من جميع الأحجام. في المشاريع الصغيرة، تميل المنهجيات إلى أن تكون عرضية وغريزية. أما في المشاريع الكبيرة، فإنها تميل إلى أن تكون دقيقة ومُخططة بعناية.

قد تكون بعض المنهجيات فضفاضة للغاية بحيث لا يدرك المبرمجون أنهم يستخدمونها. حيث يجادل عدد قليل من المبرمجين بأن المنهجيات جامدة للغاية ويقولون إنهم لن يمسوها. مع أنه قد يكون صحيحًا أن المبرمج لا يختار المنهجية بوعي، حيث أن أي أسلوب للبرمجة يشكل منهجية، بغض النظر عن الطريقة اللاواعية أو البدائية. مجرد الخروج من السرير والذهاب إلى العمل في الصباح هو منهجية بدائية وإن لم تكن إبداعية كثيرا. المبرمج الذي يصر على تجنب المنهجيات هو في الحقيقة يتجنب فقط اختيار منهجية بشكل واضح - حيث لا يمكن لأحد تجنب استخدامها كليًا.



المناهج الرسمية ليست دائمًا ممتعة، وإذا ما تم تطبيقها بشكل خاطئ، فإن نفقاتها العامة تستهلك توفيراتها الأخرى. غير أن تعقيد المشاريع الأكبر حجما يتطلب اهتمامًا أكثر وعيا بالمنهجية. حيث يتطلب بناء ناطحة سحاب منهجًا مختلفًا عن بناء بيت صغير. تعمل المشاريع البرمجية ذات الأحجام المختلفة بنفس الطريقة. في المشاريع الكبيرة، تكون الخيارات غير الواعية غير كافية للقيام بالمهمة. حيث يختار مخططو المشاريع الناجحون استراتيجياتهم للمشاريع الكبيرة بشكل صريح.

في الإطار الاجتماعي، كلما كان الحدث أكثر رسمية، كلما كانت ملابسك غير مريحة (الكعب العالي، أربطة العنق، وما إلى ذلك). في تطوير البرامج، كلما كان المشروع أكثر رسمية، كلما كان عليك إنشاء المزيد من الأوراق للتأكد من أنك قد أنجزت مهمتك. يُشير كابرز جونز إلى أن مشروعًا يتكون من 1000 سطر سيبلغ معدل حوالي 7 في المئة من مجهوده على الأعمال الورقية، في حين أن مشروعًا يبلغ طوله 100000 سطر سيبلغ معدل حوالي 26 بالمائة من جهده على الأعمال الورقية (جونز 1998).

لا يتم إنشاء هذه الأوراق للولع الشديد في كتابة الوثائق. لقد تم إنشاؤها كنتيجة مباشرة للظاهرة الموضحة في الشكل 27-1: كلما كان لديك أدمغة أناس أكثر للتنسيق، كلما زادت كمية الوثائق الرسمية التي تحتاج لتنسيقها.

أنت لا تنشئ أيًا من هذه الوثائق من أجلها هي. إن الهدف من كتابة خطة إدارة التهيئة، على سبيل المثال، ليس أن تمرن عضلاتك في الكتابة. بل الهدف من كتابتك للخطة هو إجبارك على التفكير بعناية في إدارة التهيئة وشرح خطتك لأي شخص آخر. عملية التوثيق هي أحد الآثار الجانبية الملموسة للعمل الحقيقي الذي تقوم به أثناء تخطيط وإنشاء نظام برمجي. إذا كنت تشعر أنك ترائي أثناء كتابة مستندات عامة، فهناك خطأ ما.



فيما يتعلق بالمنهجيات، "الأكثر" ليس الأفضل. يحذر باري بويم وريتشارد تيرنر في استعراضهم للمنهجيات المرنة مقابل المُقادة بالخطة، من أنك ستقوم عادة بتحسين أفضل إذا بدأت بمنهجية صغيرة ومن ثم وسّعت النطاق من أجل مشروع كبير، أفضل مما لو بدأت بمنهجية شاملة ومن ثم خفّضتها لتناسب مشروع صغير (بويم وتيرنر 2004). يتحدث بعض خبراء البرامج عن منهجيات "الوزن الخفيف" و"الوزن الثقيل"، ولكن من الناحية العملية، فإن المفتاح الأساسي هو النظر في حجم ونوع مشروعك المحدد ثم العثور على المنهجية ذات "الوزن الصحيح".

المصادر الإضافية

استخدم المصادر التالية للدراسة المتعمقة لموضوع هذا الفصل[1]:

Boehm, Barry and Richard Turner. Balancing Agility and Discipline: A Guide for the Perplexed. Boston, MA: Addison-Wesley, 2004.

بويم، وباري وريتشارد تيرنر. موازنة الرشاقة والانضباط: دليل المرتبك. بوسطن، ماساتشوستس: أديسون ويسلي، 2004. يصف بويم وتيرنر كيفية تأثير حجم المشروع على استخدام المناهج الرشيقة وتلك المُقادة بالخطة، بالإضافة إلى القضايا المرنة وتلك المُقادة بالخطة.

[1] cc2e.com/2768

Cockburn, Alistair. Agile Software Development. Boston, MA: Addison-Wesley, 2002.

كوكبرن، وأليستير. تطوير البرمجيات الرشيقة. بوسطن، ماساتشوستس: أديسون ويسلي، 2002. يناقش الفصل 4 من هذا الكتاب القضايا المتعلقة باختيار منهجيات المشروع المناسبة، بما في ذلك حجم المشروع. يقدم الفصل 6 منهجيات كريستالية ل Cockburn، وهي مناهج محددة لتطوير المشاريع ذات الأحجام المختلفة ودرجات الخطورة "الحساسية" المختلفة.

Boehm, Barry W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice Hall,1981.

بويم، باري. اقتصاد هندسة البرمجيات. إنجلوود كليفس، نيوجيرسي: برنتيس هول،1981. إن كتاب بويم هو معالجة مكثفة للتكلفة والإنتاجية وتداعيات الجودة لحجم المشروع والمتغيرات الأخرى في عملية تطوير البرمجيات. ويشمل مناقشات حول تأثير الحجم على عملية البناء وغيرها من الأنشطة. يُعد الفصل 11 تفسيرًا ممتازًا للاضطرابات التي تنتجها حجم البرمجيات. ونُشرت معلومات أخرى حول حجم المشروع في الكتاب. يحتوي كتاب بويم "تقدير تكلفة البرمجيات مع Cocomo II" لعام 2000 على المزيد من المعلومات الحديثة حول نموذج التقييم Cocomo لبويم، ولكن الكتاب الأول يقدم مناقشات خلفية أكثر عمقًا لا تزال ذات صلة.

Jones, Capers. Estimating Software Costs. New York, NY: McGraw-Hill, 1998.

جونز، كابير. تقدير تكاليف البرمجيات. نيويورك: مكجراو هيل، 1998. يحتوي هذا الكتاب على جداول ورسوم بيانية تشرح مصادر إنتاجية تطوير البرمجيات. بالنسبة لتأثير حجم المشروع على وجه التحديد، فإن كتاب جونز لعام 1986، إنتاجية البرمجة، يحتوي على مناقشة ممتازة في القسم المعنون "تأثير حجم البرنامج" في الفصل 3.

Brooks, Frederick P., Jr. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2d ed.). Reading, MA: Addison-Wesley, 1995

بروكس، فريدريك، مقالات في هندسة البرمجيات، طبعة الذكرى (النسخة الثانية). ريدينغ، أديسون ويسلي، 1995. كان بروكس مدير تطوير مشروع IBM OS/360، وهو مشروع ضخم استغرق 5000 سنة عامل. حيث يناقش القضايا الإدارية المتعلقة بالفرق الصغيرة والكبيرة ويعرض قصة حية خاصة لفرق رؤساء المبرمجين في هذه المجموعة المثيرة من المقالات.

DeGrace, Peter, and Leslie Stahl. Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms. Englewood Cliffs, NJ: Yourdon Press, 1990.

ديجريس، وبيتر، وليزلي ستال. المشاكل المؤذية والحلول الصالحة: كتالوج من النماذج الهندسية للبرمجيات الحديثة. إنجلوود كليفس، نيو جيرسي: صحافة يوردون، 1990. كما يوحي العنوان، يقترح هذا الكتاب كتالوج

من المناهج لتطوير البرمجيات. وكما هو موضح في هذا الفصل، يجب أن يتغير منهجك مع اختلاف حجم المشروع، ويوضح كل من ديجريس وستال هذه النقطة بوضوح. يناقش الفصل المعنوَن "التخفيف والتشذيب" في الفصل الخامس تخصيص عمليات تطوير البرمجيات على أساس حجم المشروع والإجرائيات الشكلية. يتضمن الكتاب وصف نماذج من وكالة ناسا ووزارة الدفاع وعددًا رائعًا من الرسوم التوضيحية.

Jones, T. Capers. "Program Quality and Programmer Productivity." IBM Technical Report TR 02.764 (January 1977): 42–78.

جونز، ت. كاريرز "جودة البرامج وإنتاجية المبرمج." تقرير IBM التقني TR 02.764 (يناير 1977): 42–78. يتوفر أيضًا في البرنامج التعليمي الخاص بشركة Jones: إنتاجية البرمجة: قضايا الثمانينات، الإصدار الثاني. لوس انجليس، كاليفورنيا: IEEE صحافة المجتمع الحاسوبي، 1986. تحتوي هذه الورقة على أول تحليل متعمق للأسباب التي تتسم بها المشاريع الكبيرة بأنماط إنفاق مختلفة عن تلك الصغيرة. إنها مناقشة شاملة للاختلافات بين المشروعات الكبيرة والصغيرة، بما في ذلك المتطلبات وتدابير ضمان الجودة. انها قديمة ولكنها لا تزال مثيرة للاهتمام.

النقاط المفتاحية

- مع زيادة حجم المشـــروع، يجب دعم الاتصـــالات. الهدف من معظم المنهجيات هو الحد من مشـــاكل الاتصالات، ويجب أن تعيش أو تموت المنهجية على أساس مزاياها كمسهل للاتصال.
- عندما تكون كل الأشياء الأخرى متساوية، فستكون الإنتاجية في مشروع كبير أقل مما هي عليه في مشروع صغير.
- عندما تكون كل الأشياء الأخرى متساوية، فالمشروع الكبير سيحوي أخطاء لكل ألف سطر من الشفرة أكثر من المشروع الصغير.
- يجب أن يتم التخطيط بعناية للأنشطة التي تعتبر بديهية في المشاريع الصغيرة في حال استخدامها في المشاريع الكبيرة. وتصبح عملية البناء أقل هيمنة مع زيادة حجم المشروع.
- يميل توســيع نطاق منهجية خفيفة إلى العمل بشــكل أفضــل من تقليص منهجية وزن ثقيل. حيث أن النهج الأكثر فعالية هو استخدام منهجية "الوزن الصحيح".

الفصل 28

إدارة البناء

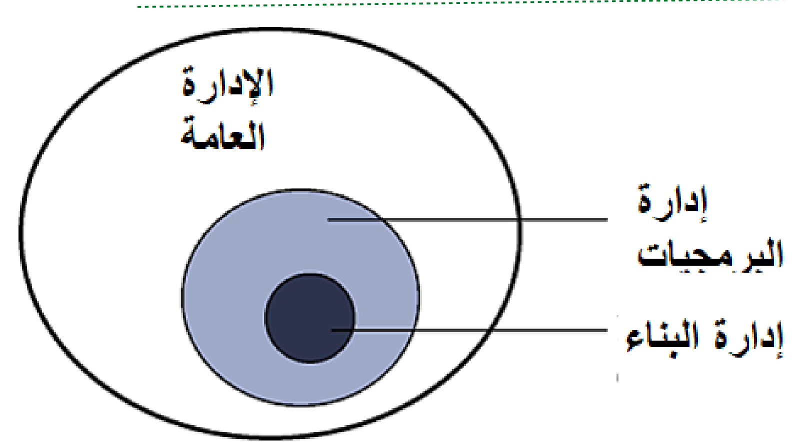
المحتويات[1]

مواضيع ذات صلة

- المتطلبات الأساسية لعملية البناء: الفصل 3
- تحديد نوع البرمجية التي تعمل عليها: القسم 2.3
- حجم البرنامج: الفصل 27
- جودة البرمجيات: الفصل 20

لقد كانت إدارة تطوير البرمجيات تحديًا هائلًا خلال العقود الماضية. كما يوضح الشكل 28-1، فإن الموضوع العام لإدارة المشاريع والبرامج يتجاوز نطاق هذا الكتاب، ولكن هذا الفصل يناقش بعض المواضيع الإدارية المحددة التي تنطبق مباشرة على عملية البناء. إذا كنت مطورًا، سيساعدك هذا الفصل على فهم القضايا التي يتعين على المدراء مراعاتها. أما إذا كنت مديرًا، فسيساعدك هذا الفصل على فهم كيفية نظر الإدارة إلى المطورين، وكذلك كيفية إدارة عملية البناء بفعالية. ونظرًا لأن هذا الفصل يغطي مجموعة واسعة من المواضيع، فإن العديد من أقسامه تصف أيضًا المكان الذي يمكنك الانتقال إليه للحصول على مزيد من المعلومات.

[1] cc2e.com/2836



الشكل 28-1 يغطي هذا الفصل مواضيع إدارة البرمجيات المتعلقة بالبناء.

إذا كنت مهتمًا بإدارة البرمجيات، فاحرص على قراءة القسم 2.3، "تحديد نوع البرمجيات التي تعمل عليها"، لفهم الفرق بين النهج التعاقبية التقليدية للتطوير والنهج العصرية التكرارية. تأكد أيضًا من قراءة الفصل 20، "المنظر الطبيعي لجودة البرمجيات"، والفصل 27، "كيف يؤثر حجم البرنامج على عملية البناء". حيث تؤثر أهداف الجودة وحجم المشروع بشكل كبير على كيفية إدارة مشروع برمجي مُحدّد.

28. 1 تشجيع الكتابة الجيدة للشفرة

نظرًا لأن الشفرة هي الناتج الأساسي من عملية البناء، فإن السؤال الرئيسي في إدارة البناء هو "كيف تشجع ممارسات كتابة الشفرة الجيدة؟". بشكل عام، لا يعد وضع مجموعة صارمة من المعايير التقنية من قبل الإدارة فكرة جيدة. يميل المبرمجون إلى النظر إلى المدراء على أنهم في مستوى أدنى من التطور التقني، في مكان ما بين الكائنات أحادية الخلية والماموث الصوفي الذي انقرض خلال العصر الجليدي، وإذا كانت هناك معايير برمجة، فيحتاج المبرمجون البدء بمحاولة معرفتها.

إن كان شخص ما في مشروع سيحدد المعايير، فلابد من تحديد مسؤول عن الهيكلة محترم للمعايير بدلاً من المدير. تعمل مشروعات البرمجيات أكثر على "التسلسل الهرمي للخبرة" أكثر من "التسلسل الهرمي للسلطة". إذا تم اعتبار المُهيكل قائدًا للفكر في المشروع، فسيتبع فريق المشروع عمومًا المعايير التي يضعها ذلك الشخص.

إذا اخترت هذا النهج، تأكد من احترام المُهيكل حقًا. في بعض الأحيان، يكون مهيكل المشروع مجرد شخص كبير قد مضى وقتًا طويلًا جدًا، بعيدًا عن مشكلات كتابة شفرة الإنتاج. حيث يستاء المبرمجون من هذا النوع من "المهيكل" الذي يحدد معايير بعيدة عن العمل الذي يقومون به.

اعتبارات في وضع المعايير

إن المعايير أكثر فائدة في بعض المنظمات من بعضها الآخر. حيث يُرحب بعض المطورين بالمعايير لأنها تقلل التباين العشوائي في المشروع. فإذا قاومت مجموعتك اعتماد معايير صارمة، فكر في بعض البدائل: إرشادات مرنة، مجموعة من الاقتراحات بدلًا من الإرشادات، أو مجموعة من الأمثلة التي تجسد أفضل الممارسات.

تقنيات لتشجيع كتابة الشفرة الجيدة

يصف هذا القسم عدة تقنيات لتحقيق ممارسات كتابة شفرة جيدة، أقل ثقلًا من وضع معايير صارمة لكتابة الشفرة:

قم بتعيين شخصين لكل جزء من المشروع[1]. إذا كان على شخصين العمل على كل سطر من الشفرة، فستضمن أن شخصين على الأقل يعتقدان أنه يعمل وقابل للقراءة. يمكن أن تتراوح آليات التعاون بين شخصين من البرمجة الثنائية لتوجيه أزواج المتدربين إلى مراجعات نظام الأصدقاء.

راجع كل سطر من الشفرة[2]. تتضمن مراجعة التعليمات البرمجية عادةً المبرمج ومراجعين اثنين على الأقل. وهذا يعني أن ثلاثة أشخاص على الأقل يقرؤون كل سطر من الشفرة. اسم آخر لمراجعة النظير هو " تأثير الأقران ". فبالإضافة إلى توفير شبكة أمان في حالة مغادرة المبرمج الأصلي للمشروع، تقوم المراجعات بتحسين جودة الشفرة لأن المبرمج يعرف أن الشفرة سوف يقرأها الآخرون. حتى إذا لم تقم شركتك بإنشاء معايير صريحة لكتابة الشفرة، فإن المراجعات تقدم طريقة دقيقة للتحرك نحو معيار جماعي لكتابة الشفرة– حيث تتخذ القرارات من قبل المجموعة أثناء المراجعات، وبمرور الوقت تستمد المجموعة معاييرها الخاصة.

اطلب توقيع الشفرة. في مجالات أخرى، يتم اعتماد الرسومات التقنية وتوقيعها من قبل المهندس الإداري. يعني التوقيع أنه وفقًا لأفضل معرفة للمهندس، تكون الرسومات كفأة تقنيًا وخالية من الأخطاء. تتعامل بعض الشركات مع الشفرة بنفس الطريقة. قبل اعتبار الشفرة كاملة، يجب على كبار الموظفين التقنيين التوقيع على قائمة التعليمات البرمجية.

وجّه أمثلة كتابة شفرة جيدة للمراجعة. إن جزء كبير من الإدارة الجيدة هو توصيل أهدافك بوضوح. إحدى الطرق لإيصال أهدافك هي تعميم شفرة جيدة للمبرمجين أو نشرها للعرض العام. عند القيام بذلك، فإنك تقدم مثالًا واضحًا على الجودة التي تهدف لتحقيقها. وبالمثل، يمكن أن يتكون دليل معايير كتابة الشفرة أساسًا من

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن البرمجة الثنائية، انظر القسم 21.2 " البرمجة الثنائية ".

هامش: نظام الأصدقاء هو إجراء يعمل فيه شخصان، "الرفقاء"، كوحدة واحدة حتى يتمكنوا من مراقبة ومساعدة بعضهم البعض.

[2] **إشارة مرجعية**: لمزيد من التفاصيل عن المراجعات، انظر الفصل 21.3 "التفحصات الرسمية"، والقسم 21.4 " الأنواع الأخرى من تطبيقات التطوير التعاوني ".

مجموعة من "أفضل الشفرات". إن تحديد قوائم معينة كـ "أفضل" يُعد مثالًا يحذو حذوه الآخرون. مثل هذا الدليل أسهل للتحديث من دليل معايير اللغة الإنجليزية، ويقدم دون عناء خفايا في أسلوب كتابة الشفرة والتي يصعب ضبطها نقطة تلو الأخرى في الأوصاف النثرية.

أكّد أن قوائم الشفرة هي ممتلكات عامة[1]. يشعر المبرمجون أحيانًا أن الشفرة التي كتبوها هي "شفرتهم"، كما لو كانت ملكية خاصة. على الرغم من أنها تمثل نتيجة لعملهم، إلا أن الشفرة هي جزء من المشروع ويجب أن تكون متاحه مجانًا لأي شخص آخر في المشروع يحتاج إليها. يجب أن يراها الآخرون أثناء المراجعات والصيانة، حتى لو لم يحدث ذلك في أي وقت آخر.

واحد من أكثر المشاريع الناجحة على الإطلاق المذكورة تتكون من 83000 سطر من الشفرة أُنجز في 11 سنة عمل. وتم اكتشاف خطأ واحد فقط أدى إلى فشل النظام في أول 13 شهر من التشغيل. يكون هذا الإنجاز أكثر دراماتيكية عندما تدرك أن المشروع تم الانتهاء منه في أواخر الستينات، دون تنسيق عبر الإنترنت أو تصحيح تفاعلي للأخطاء. لا تزال الإنتاجية في المشروع - 7500 سطر من الشفرة لكل سنة عمل في أواخر الستينات- مثيرة للإعجاب وفقًا لمعايير اليوم. وأفاد كبير مبرمجي المشروع أن أحد مفاتيح نجاح المشروع هو تحديد جميع المسارات الحاسوبية (الخاطئة وغيرها) كممتلكات عامة وليست خاصة (بيكر وميلز 1973). امتدت هذه الفكرة إلى سياقات حديثة، بما في ذلك البرمجة مفتوحة المصدر (ريمون 2000)، وإلى فكرة البرمجة القصوى "المتطرفة" للملكية الجماعية (بيك 2000)، وفي سياقات أخرى أيضا.

كافئ الشفرة الجيدة. استخدم نظام المكافآت لمؤسستك لتعزيز ممارسات كتابة الشفرة الجيدة. ضع هذه الاعتبارات في ذهنك أثناء تطوير نظام المكافئة:

- يجب أن تكون الجائزة شـــيئًا يريده المبرمج. حيث يجد العديد من المبرمجين مكافآت من نوع "ياه، جيد" مستهجنة، خاصة عندما تأتي من مدراء غير تقنيين.
- يجب أن تكون الشفرة التي تحصل على الجائزة جيدة للغاية. إن أعطيت جائزة لمبرمج يعرف الجميع أن عمله ســيء، فأنت تبدو مثل هومر ســيمبســون وهو يحاول تشــغيل مفاعل نووي. لا يهم إن كان المبرمج يبدي سلوكً تعاونيًا أو أنه يأتي دائمًا للعمل في الوقت المناسب. حيث ستفقد مصداقيتك إذا كانت مكافأتك لا تتطابق مع المزايا التقنية للحالة. إذا لم تكن ماهرًا تقنيًا بما يكفي للتحكيم على

[1] **إشارة مرجعية**: قسم كبير من البرمجة هو ربط عملك مع عمل الآخرين. لمزيد من التفاصيل، انظر القسم 33. 5 والقسم 34. 3.

الشـــفرة الجيدة، فلا تفعل ذلك! لا تقوم بالمكافئة على الإطلاق، أو اســـمح لفريقك باختيار مُســـتلم المكافئة.

معيار واحد سهل. إذا كنت تدير مشروعًا برمجيًا ولديك خلفية برمجية، فإن الأسلوب السهل والفعّال لاستنباط العمل الجيد هو أن تقول "يجب أن أكون قادرًا على قراءة وفهم أي شفرة مكتوبة للمشروع." ذلك أن المدير ليس أمهر شخص تقنيا، ويمكن وجود ميزة في ذلك وهي إمكانية تثبيط الشفرة المعقدة أو "الذكية".

دور هذا الكتاب

معظم أجزاء هذا الكتاب هي مناقشة لممارسات البرمجة الجيدة. لم يُقصد منه أن يُستخدم لتبرير المعايير الصارمة، بل قُصد منه حتى أقل من أن يستخدم كمجموعة من المعايير الصارمة. استخدم هذا الكتاب كأساس للمناقشة، وكمصدر لممارسات البرمجة الجيدة، ولتحديد الممارسات التي يمكن أن تكون مفيدة في بيئتك.

2.28 إدارة الإعداد "التكوين"

مشروع البرمجيات هو مشروع ديناميكي. حيث تتغير الشفرة، ويتغير التصميم، وتتغير المتطلبات. وأكثر من ذلك، تؤدي التغييرات في المتطلبات إلى مزيد من التغييرات في التصميم، وتؤدي التغييرات في التصميم إلى المزيد من التغييرات في الشفرة وحالات الاختبار.

ما هي إدارة الإعداد "التكوين"؟

إدارة الإعداد هي ممارسة تحديد أدوات المشروع والتعامل مع التغييرات بطريقة منتظمة بحيث يتمكن النظام من الحفاظ على تكامله مع مرور الوقت. وله اسم آخر وهو "التحكم بالتغيير". ويشمل ذلك تقنيات لتقييم التغييرات المقترحة، وتتبّع التغييرات، وحفظ نسخ من النظام كما كانت في نقاط زمنية مختلفة.

إذا كنت لا تتحكم في التغييرات التي تطرأ على المتطلبات، فيمكنك في نهاية الأمر كتابة شفرة لأجزاء من النظام يتم التخلص منها في النهاية. وقد يتم حينها كتابة شفرة غير متوافقة مع أجزاء جديدة للنظام. وقد لا تكتشف العديد من حالات عدم التوافق حتى وقت التكامل، والذي سيصبح وقتًا لتوجيه أصابع الاتهام لأنه حقيقة لا يوجد أحد يعرف ما الذي يحدث.

إذا لم يتم التحكم في التغييرات التي يتم إجراؤها على الشفرة، فقد تُغيّر إجرائية يُغيرها شخص آخر في الوقت نفسه؛ عندها سيكون نجاح الدمج بين التغييرات الخاصة بك مع تغييرات الآخرين إشكالية. يمكن أن تؤدي التغييرات غير المضبوطة للشفرة إلى جعل الشفرة تبدو مُختبرة أكثر مما هي عليه في الحقيقة. ربما يكون الإصدار الذي تم اختباره هو الإصدار القديم الذي لم يتغير، وربما لم يتم اختبار الإصدار المُعدّل. فبدون التحكم

الجيد في التغيير، قد تجري تغييرات على الإجرائية، وتعثر على أخطاء جديدة، ولن تكون قادرا على التراجع إلى الإجرائية العاملة القديمة.

وتستمر المشاكل إلى أجل غير مُسمى. إذا لم تُعالج التغييرات بشكل منتظم، فأنت تمشي خطوات عشوائية في الضباب بدلًا من الانتقال مباشرةً نحو وجهة واضحة. بدون التحكم في التغيير الجيد، عندها بدلًا من تطوير الشفرة، فإنك تهدر وقتك هباءً. إن إدارة الإعداد تساعدك على استخدام وقتك بفعالية.

على الرغم من الحاجة الواضحة لإدارة الإعداد، فقد تجنبه العديد من المبرمجين لعقود. وجدت دراسة استقصائية أجريت منذ أكثر من 20 عامًا أن أكثر من ثلث المبرمجين لم يكونوا على دراية بالفكرة (بيك و بيركنز 1983)، وهناك القليل من المؤشرات التي تشير إلى تغير ذلك. وجدت دراسة حديثة **لمعهد هندسة البرمجيات** أن المؤسسات التي تستخدم ممارسات تطوير البرمجيات غير الرسمية، أقل من 20 في المئة منها لديها إدارة تكوين مناسبة (SEI 2003).

لم يخترع المبرمجون إدارة الإعداد، ولكن نظرًا لأن مشاريع البرمجة متقلبة جدًا، فإنها مفيدة بشكل خاص للمبرمجين. إن إدارة الإعداد المُطبقة على مشاريع البرمجيات تُسمى عادةً "إدارة إعداد البرمجيات" (SCM). تُركز إدارة إعداد البرمجيات SCM على متطلبات البرنامج وشفرة المصدر والوثائق وبيانات الاختبار.

إن المشكلة الشاملة في إدارة إعداد البرمجيات SCM هي التحكم الزائد. فكما إن أضمن طريقة لوقف حوادث السيارات هي منع الجميع من القيادة، فهناك طريقة أكيدة لمنع مشاكل تطوير البرمجيات هي إيقاف جميع عمليات تطوير البرمجيات. على الرغم من أن هذه إحدى طرق التحكم في التغييرات، إلا أنها طريقة مرعبة لتطوير البرمجية. عليك أن تخطط بعناية لإدارة إعداد البرمجيات SCM بحيث تكون مُساعدَة لك بدلاً من ان تكون عقبة تعترض نجاحك.

في مشروع صغير، شخص واحد، يمكنك القيام بعمل جيد مع عدم وجود إدارة إعداد البرمجيات SCM بتجاوز التخطيط لعمليات النسخ الاحتياطي الدورية غير الرسمية[1]. ومع ذلك، لا تزال إدارة الإعداد مفيدة (وفي الواقع، لقد استخدمت إدارة الإعداد في كتابة هذا الكتاب). أما في مشروع كبير يتألف من 50 شخصًا، ستحتاج على الأرجح إلى خطة إدارة إعداد البرمجيات SCM كاملة، بما في ذلك إجراءات رسمية لعمليات النسخ الاحتياطي، والتحكم بتغيّر المتطلبات والتصميم، والتحكم بالوثائق، وشفرة المصدر، والمحتوى، وحالات الاختبار، وغيرها من أجزاء المشروع. إذا كان مشروعك ليس كبيرًا جدًا ولا صغيرًا للغاية، فسيتعين عليك الاستقرار على شكل

[1] **إشارة مرجعية**: لمزيد من التفاصيل عن تأثير حجم المشروع على عملية البناء، انظر الفصل 27 "كيف يؤثر حجم المشروع على عملية البناء".

معين في مكان ما بين الطرفين. تصف الأقسام الفرعية التالية بعض الخيارات في تنفيذ إدارة إعداد البرمجيات SCM.

تغييرات التصميم و المتطلبات[1]

أثناء عملية التطوير، من المفترض أن تكون مفعما بالأفكار حول كيفية تحسين النظام. إذا نفذت كل تغيير أثناء حدوثه، فستجد نفسك قريبًا تسير في حلقة مفرغة للبرمجية - مع أن النظام سيتغير، إلا أنه لن يقترب من الاكتمال. فيما يلي بعض الإرشادات للتحكم في تغييرات التصميم:

اتّبع إجرائية منظمة في التحكم بالتغيير. كما أشار القسم 3.4، فإن إجرائية تحكم بالتغيير مُنظمة هي هبة من السماء عندما يكون لديك الكثير من متطلبات التغيير. من خلال إنشاء إجرائية منظمة، فإنك توضح أنه سيتم النظر في التغييرات في أفضل سياق للمشروع عمومًا.

تعامل مع طلبات التغيير في مجموعات. من المغري تنفيذ تغييرات سهلة عند ظهور الأفكار. المشكلة في التعامل مع التغييرات بهذه الطريقة هي أن التغييرات الجيدة يمكن أن تضيع. إذا فكرت في تغيير بسيط للطريق بنسبة 25 في المئة خلال المشروع وأنت ضمن الجدول الزمني، فستقوم بإجراء التغيير. وإذا فكرت في تغيير بسيط آخر للطريق بنسبة 50 في المئة خلال المشروع وكنت متأخراً بالفعل، فلن تفعل ذلك. عندما يبدأ الوقت بالنفاد في نهاية المشروع، لا يهم أن يكون التغيير الثاني جيدًا بمقدار 10 أضعاف المستوى الأول، ولن تكون في وضع يسمح لك بإجراء أي تغييرات ثانوية غير الأساسية. يمكن لبعض من أفضل التغييرات التسرب من خلال الشقوق لمجرد أنك فكرت فيها في وقت متأخر بدلًا من وقت مُبكر.

حل هذه المشكلة هو كتابة كل الأفكار والمقترحات، بغض النظر عن مدى سهولة تنفيذها، والاحتفاظ بها لحين توفر الوقت الكافي للعمل عليها. بعد ذلك، قم بدراستها كمجموعة، واختر منها الأكثر فائدة.

قدّر تكلفة كل تغيير. في حال كان عميلك، أو رئيسك، أو أنت تميلون إلى تغيير النظام، فقم بتقدير الوقت الذي سيستغرقه لإجراء التغيير، بما في ذلك مراجعة شفرة التغيير وإعادة اختبار النظام بأكمله. قم بتضمين الوقت التقديري الخاص بك للتعامل مع تأثير تموج التغيير خلال المتطلبات للتصميم وللشفرة وللاختبار وللتغييرات في وثائق المستخدم. دع جميع الأطراف المهتمة تعرف أن البرنامج متشابك بشكل معقد وأن تقدير الوقت ضروري حتى وإن بدا التغيير صغيرًا للوهلة الأولى.

[1] **إشارة مرجعية**: بعض نهج التطوير تدعم التغيرات بشكل أفضل من بعضها الآخر. لمزيد من التفاصيل، انظر القسم 2.3 "تحديد نوع البرمجيات التي تعمل عليها".

بغض النظر عن مدى شعورك بالتفاؤل عندما يتم اقتراح التغيير، عليك الامتناع عن إعطاء تقدير غير مناسب. غالبًا ما يتم وصف هذه التقديرات عن طريق عامل خطأ 2 أو أكثر.

كن حذرًا من أحجام التغيير المرتفعة[1]. في حين أن درجة معينة من التغيير أمر لا بد منه، إلا أن عددًا كبيرًا من طلبات التغيير يمثل علامة تحذير رئيسية مفادها أن المتطلبات أو البنية أو التصميمات ذات المستوى الأعلى لم تكن جيدة بما يكفي لدعم البناء الفعّال. قد يبدو النسخ الاحتياطي للعمل على المتطلبات أو البنية باهظ الثمن، ولكنه لن يكون مكلفًا كتكلفة إنشاء البرنامج أكثر من مرة أو التخلص من شفرة الميزات التي لم تكن بحاجة إليها حقًا.

أنشئ مجلس التحكم في التغيير أو ما يعادله بطريقة منطقية لمشروعك. وظيفة مجلس التحكم في التغيير هي فصل الجيد عن الرديء "القمح عن القشر" في طلبات التغيير. أي شخص يريد اقتراح تغيير يقدم طلب التغيير إلى مجلس التحكم في التغيير. يشير مصطلح "طلب التغيير" إلى أي طلب من شأنه تغيير البرنامج: فكرة لميزة جديدة، أو تغيير في ميزة موجودة، أو "تبليغ عن خطأ" قد يكون حقيقيا أو لا يكون، وما إلى ذلك. حيث يجتمع المجلس دوريًا لمراجعة التغييرات المقترحة. حيث يمكنه أن يوافق على كل تغيير أو يعارضه أو يرفضه. تعتبر مجالس التحكم في التغيير من أفضل الممارسات لتحديد الأولويات والتحكم في تغييرات المتطلبات؛ ومع ذلك، فإنها لا تزال غير شائعة إلى حد ما في الإعدادات التجارية (جونز 1998، جونز 2000).

راقب البيروقراطية، لكن لا تدع الخوف من البيروقراطية يحول دون السيطرة الفعالة على التغيير. يعد عدم وجود السيطرة على التغيير المنضبط واحدة من أكبر المشاكل الإدارية التي تواجه صناعة البرمجيات اليوم. نسبة كبيرة من المشاريع التي يُتوقع تأخرها ستنجز بالواقع في الموعد المحدد إن حسبت حساب تأثير التغييرات غير المُتتبَّعة ولكن المتفق عليها. تسمح الرقابة السيئة للتغيير بتغييرات متأخرة عن السجلات، مما يقوض الرؤية المستقبلية، والقدرة على التنبؤ بعيد المدى، وتخطيط المشروع، وإدارة المخاطر بشكل خاص، وإدارة المشاريع بشكل عام.

يميل التحكم في التغيير إلى الانجراف نحو البيروقراطية، لذلك من المهم البحث عن طرق لتبسيط عملية التحكم في التغيير. إذا كنت تفضل عدم استخدام طلبات التغيير التقليدية، فقم بإعداد عنوان بريد إلكتروني بديل باسم "ChangeBoard" واطلب من الأشخاص إرسال طلبات التغيير إلى عنوان البريد الإلكتروني. أو اطلب من الناس تقديم مقترحات التغيير بشكل تفاعلي في اجتماع مجلس إدارة التغيير. أحد النهج القوية بشكل خاص تتمثل بتسجيل طلبات التغيير على أنها عيوب في برنامج تتبع العيوب لديك. حيث سيصنف

1 إشارة مرجعية: لمنظور آخر للتعامل مع التغييرات انظر "التعامل مع تغييرات المتطلبات أثناء البناء" في القسم 4.3. وللحصول على نصيحة حول التعامل الآمن مع تغييرات الشفرة عند حدوثها، انظر الفصل 24 " إعادة التصنيع "

الأصوليون "الذين يصرون على الالتزام المطلق بالقواعد" هذه التغييرات كـ "عيوب المتطلبات"، أو يمكنك تصنيفها كتغييرات بدلاً من عيوب.

ويمكنك تنفيذ التغير عن طريق مجلس التحكم في التغيير نفسه بشكل رسمي، أو يمكنك تحديد فريق تخطيط المنتجات أو هيئة حرب تحمل المسؤوليات التقليدية لمجلس التحكم في التغيير. أو يمكنك تحديد شخص واحد ليكون قيصر التغيير. لكن بغض النظر عن التسمية، نَفّذه!

ألحظ من وقت لآخر المشاريع التي تعاني من تطبيقات التحكم في التغيير. لكني أرى في بعض الأحيان عشرة أضعاف المشاريع التي تعاني من عدم وجود أي تحكم حقيقي في التغيير. إن عنصر التحكم في التغيير هو المهم، لذلك لا تدع الخوف من البيروقراطية يمنعك من تحقيق فوائده العديدة.

التغييرات في شفرة البرمجيات

مشكلة أخرى في إدارة الإعداد هي التحكم في شفرة المصدر. إذا غيّرت الشفرة وظهرت أخطاء جديدة تبدو أنها غير مرتبطة بالتغيير الذي أجريته، فربما تريد مقارنة الإصدار الجديد من الشفرة بالإصدار القديم في بحثك عن مصدر الخطأ. إذا لم يخبرك ذلك بأي شيء، فقد تحتاج إلى النظر إلى الإصدار الأقدم. هذا النوع من الرحلات عبر التاريخ يكون سهلًا إذا كان لديك أدوات للتحكم في الإصدار والتي تتبع إصدارات متعددة من شفرة المصدر.



برمجية التحكم في الإصدار. تعمل البرمجية الجيدة للتحكم في الإصدار بسهولة بحيث لا تكاد تلاحظ أنك تستخدمها. إنها مفيدة بشكل خاص في مشاريع الفريق. حيث يعمل أحد أساليب التحكم في الإصدار على قفل ملفات المصدر بحيث يمكن لشخص واحد فقط تعديل ملف في كل مرة. عادةً، عندما تحتاج إلى العمل على شفرة المصدر في ملف معين، يمكنك فحص الملف خارج إطار التحكم بالإصدار. إذا كان شخص آخر يفحص الملف في ذلك الوقت، فسيتم إشعارك بأنه لا يمكنك فحصه. وعندما يمكنك فحص الملف، فإنك تعمل عليه تمامًا كما تفعل دون التحكم في الإصدار إلى أن تصبح جاهزًا لفحصه..

يسمح أسلوب آخر لعدة أشخاص بالعمل في وقت واحد على الملفات ويتعامل مع مسألة دمج التغييرات عند فحص الشفرة. في كلتا الحالتين، عندما تقوم بفحص الملف، يسألك التحكم في الإصدار عن سبب تغييره، وعندها عليك أن تكتب سببًا.

ستحصل على العديد من الفوائد الكبيرة لهذا الاستثمار المتواضع في الجهد:

- لن تدعس لأحد على طرف من خلال العمل على ملف بينما يعمل عليه شخص آخر أيضًا (أو على الأقل ستعرف عنه إذا فعلت ذلك).

- يمكنك بسـهولة تحديث نسـخك من جميع ملفات المشـروع إلى الإصـدارات الحالية، عادة عن طريق إصدار أمر واحد.
- يمكنك الرجوع إلى أي إصدار من أي ملف تم فحصه في التحكم في الإصدار.
- يمكنك الحصول على قائمة بالتغييرات التي تم إجراؤها على أي إصدار من أي ملف.

لا داعي للقلق بشأن النسخ الاحتياطية الشخصية نظرًا لأن نسخة التحكم في الإصدار هي شبكة آمنة.

لا غنى عن التحكم في الإصدار في مشاريع الفريق. وتصبح أكثر قوة عندما يتم دمج التحكم في الإصدار، وتتبع العيوب، وإدارة التغيير مع بعضهم البعض. وجد قسم تطبيقات مايكروسوفت في أداة التحكم في الإصدارات الخاصة به "ميزة تنافسية كبرى" (مور 1992).

إصدارات الأداة

بالنسبة لبعض أنواع المشاريع، قد يكون من الضروري أن تكون قادرًا على إعادة بناء البيئة الدقيقة المستخدمة لإنشاء كل إصدار محدد من البرنامج، بما في ذلك المترجمات، والرابطات، ومكتبات الشفرة، وما إلى ذلك. في هذه الحالة، يجب عليك وضع كل تلك الأدوات في التحكم في الإصدار أيضًا.

إعدادات الجهاز

شهدت العديد من الشركات (بما في ذلك شركتي) نتائج جيدة من إنشاء إعدادات موحدة لتطوير الجهاز. حيث يتم إنشاء صورة قرص من محطة عمل مطور قياسية، بما في ذلك جميع أدوات المطور الشائعة والتطبيقات المكتبية وما إلى ذلك. ويتم تحميل هذه الصورة على جهاز كل مطور. حيث يساعد وجود إعدادات قياسية على تجنب مجموعة من المشكلات المرتبطة بإعدادات تكوين مختلفة قليلًا، وإصدارات مختلفة من الأدوات المستخدمة، وما إلى ذلك. كما تسهل صورة القرص الموحدة بشكل كبير إعداد أجهزة جديدة مقارنةً بضرورة تثبيت كل برمجية على حدة.

الخطة الاحتياطية

لا تُعد الخطة الاحتياطية مفهومًا جديدًا دراماتيكيًا؛ إنها فكرة دعم عملك بشكل دوري. مثلًا إذا كنت تكتب كتابًا يدويًا، فلن تترك الصفحات في كومة على الشرفة. إذا فعلت ذلك، فقد يتبللون بالمطر أو يذهبون مع الريح، أو قد يستعيرهم كلب جارك لقراءة القليل قبل النوم. بل ستضعهم في مكان آمن. البرمجيات أقل قابلية للمس، لذلك من الأسهل نسيان أن لديك شيئًا ذا قيمة هائلة على جهاز واحد.

حيث يمكن أن تحدث أشياء كثيرة للبيانات المحوسبة: يمكن أن يعطب القرص؛ أو يمكنك أنت أو أي شخص آخر حذف الملفات الرئيسية عن طريق الخطأ؛ أو يمكن لموظف غاضب تخريب جهازك؛ أو قد تفقد جهاز الحاسوب نتيجة سرقة أو فيضان أو حريق. لذلك يجب اتخاذ خطوات لحماية عملك. يجب أن تتضمن خطة

النسخ الاحتياطي الخاصة بك عمل نسخ احتياطية على أساس دوري ونقل النسخ الاحتياطية الدورية إلى التخزين الخارجي، ويجب أن يشمل هذا جميع المواد الهامة في مشروعك- المستندات والرسومات والملاحظات - بالإضافة إلى شفرة المصدر.

أحد الجوانب التي غالباً ما يتم تجاهلها في إنشاء خطة النسخ الاحتياطي هو اختبار لإجراءات النسخ الاحتياطي. حاول إجراء استعادة في مرحلة ما للتأكد من احتواء النسخ الاحتياطي على كل ما تحتاج إليه وأن الاسترداد يعمل بشكل صحيح.

عند الانتهاء من مشروع ما، قم بعمل أرشيف للمشروع. احفظ نسخة من كل شيء: شفرة المصدر، المترجمات، الأدوات، المتطلبات، التصميم، الوثائق - كل ما تحتاجه لإعادة إنشاء المنتج. واحتفظ بكل شيء في مكان آمن.

لائحة اختبار: إدارة الإعداد بشكل عام[1]

- هل تم تصميم خطة إدارة إعداد البرمجيات لمساعدة المبرمجين وتقليل النفقات العامة؟
- هل يتجنب منهج إدارة إعداد البرمجيات لديك التحكم الزائد بالمشروع؟
- هل تقوم بتجميع طلبات التغيير، إما من خلال و سائل غير ر سمية (مثل قائمة التغييرات المُعلقة) أو من خلال نهج أكثر انتظامًا (مثل مجلس التحكم في التغيير)؟
- هل تقوم بشكل منهجي بتقدير التكلفة، والجدول الزمني، وتأثير الجودة لكل تغيير مُقترح؟
- هل ترى التغييرات الرئيسة بمثابة تحذير بأن تطوير المتطلبات لم يكتمل بعد؟

الأدوات

- هل تستخدم برمجية التحكم في الإصدار لتسهيل إدارة الإعداد؟
- هل تستخدم برمجية التحكم في الإصدار لتقليل مشاكل التنسيق المتعلقة بالعمل في الفرق؟

النسخ الاحتياطي

- هل تقوم بعمل نسخ احتياطية لكافة مواد المشروع بشكل دوري؟
- هل يتم نقل النسخ الاحتياطية للمشروع إلى موقع تخزين خارجي بشكل دوري؟
- هل يتم نسـخ جميع المواد احتياطيًا، بما في ذلك شـفرة المصـدر، والمسـتندات، والرسـومات، والملاحظات المُهمة؟
- هل قمت باختبار إجراء استرداد النسخ الاحتياطي؟

[1] cc2e.com/2843

مصادر إضافية عن إدارة الإعداد

نظرًا لأن هذا الكتاب يتعلق بعملية البناء، فقد ركز هذا القسم على التحكم في التغيير من وجهة نظر البناء[1]. لكن التغييرات تؤثر في المشاريع على جميع المستويات، ويجب على الاستراتيجية الشاملة للتحكم في التغيير أن تحذو حذوها.

Hass, Anne Mette Jonassen. Configuration Management Principles and Practices. Boston, MA: Addison-Wesley, 2003.

هاس، آن ميتي يوناسن. مبادئ إدارة الإعداد والممارسات العملية. بوسطن، ماساتشوستس: أديسون ويسلي، 2003. يقدم هذا الكتاب عرضًا كبيرًا لإدارة إعداد البرمجيات وتفاصيل عملية حول كيفية دمجها في عملية تطوير البرمجيات. وهو يركز على إدارة عناصر الإعداد والتحكم فيها.

Berczuk, Stephen P. and Brad Appleton. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston, MA: Addison-Wesley, 2003.

بيرسزوك، ستيفن ب. وبراد أبلتون. أنماط إدارة إعداد البرمجيات: العمل الجماعي الفعّال، التكامل العملي. بوسطن، ماساتشوستس: أديسون ويسلي، 2003. مثل كتاب هاس، يقدم هذا الكتاب نظرة عامة عن إدارة إعداد البرمجيات وهو كتاب عملي. وهو يكمل كتاب هاس من خلال توفير إرشادات عملية تسمح لفرق المطورين بعزل وتنسيق عملهم.

SPMN. Little Book of Configuration Management. Arlington, VA: Software Program Managers Network, 1998.

SPMN. كتاب صغير لإدارة الإعداد. أرلينغتون، فرجينيا: شبكة برامج مدراء البرامج، 1998[2]. هذا الكتيب عبارة عن مقدمة لأنشطة إدارة الإعداد ويحدد عوامل النجاح الحاسمة. وهي متاحة كتحميل مجاني من موقع SPMN على www.spmn.com/products_guidebooks.html.

Bays, Michael. Software Release Methodology. Englewood Cliffs, NJ: Prentice Hall, 1999.

[1] cc2e.com/2850

[2] cc2e.com/2857

بايز، مايكل. منهجية تحرير البرمجيات. إنجلوود كليفس، نيوجيرسي: برنتيس هول،1999. يناقش هذا الكتاب إدارة إعداد البرمجيات مع التركيز على تحرير البرمجيات داخل مرحلة الإنتاج.

Bersoff, Edward H., and Alan M. Davis. "Impacts of Life Cycle Models on Software Configuration Management." Communications of the ACM 34, no. 8 (August 1991):104–118.

بيرسوف، إدوارد، وآلان، وديفيس. "آثار نماذج دورة الحياة على إدارة إعداد البرمجيات." اتصالات ACM 34، رقم. 8 (آب / أغسطس 1991): 104-118. توضح هذه المقالة كيف يتأثر إدارة إعداد البرمجيات SCM بالمناهج الحديثة لتطوير البرامج، وخاصةً نُهج النماذج الأولية. تنطبق هذه المقالة بشكل خاص في البيئات التي تستخدم ممارسات التطوير السريع.

28. 3 تقدير الجدول الزمني لعملية البناء

بيانات مثبتة

تعد إدارة مشروع البرمجيات أحد التحديات الهائلة في القرن الحادي والعشرين، ويعد تقدير حجم المشروع والجهد المطلوب لإكماله أحد أكثر الجوانب صعوبةً في إدارة مشروعات البرمجيات. حيث أن متوسط المشروع البرمجيات الكبيرة متأخّر بسنة واحدة و100 في المئة أعلى من الميزانية (مجموعة ستانديش 1994، جونز 1997، جونسون 1999). على المستوى الفردي، وجدت الدراسات الاستقصائية المقدرة مقابل الجداول الفعلية أن تقديرات المطورين تميل إلى وجود عامل تفاؤل يتراوح بين 20 و 30 في المئة (فان جنوتشن 1991). وهذا له علاقة بضعف تقدير الحجم والجهد كما هو الحال مع جهود التطوير الضعيفة. يوضح هذا القسم المشكلات التي ينطوي عليها تقدير المشروعات البرمجية ويشير إلى مكان البحث عن مزيد من المعلومات.

مناهج التقدير

يمكنك تقدير حجم المشروع والجهد المطلوب لإكماله باستخدام أي من الطرق العديدة[1]:

- استخدم تقدير البرمجيات.
- استخدم منهجًا حسابيًا مثل كوكومو 2 (Cocomo II)، نموذج تقدير باري بويم، (بويم وآخرون 2000).
- اطلب من خبراء التقدير الخارجيين تقدير المشروع.
- أعقد اجتماع من أجل التقديرات.
- قم بتقدير أجزاء من المشروع، وثم قم بإضافة الأجزاء معًا.

[1] **قراءة متعمقة**: للمزيد حول تقنيات لتقدير الجداول الزمنية، انظر الفصل 8 من التطوير السريع (ماكونيل 1996)، وتقدير تكلفة البرمجيات مع Cocomo II (بويم وآخرون. 2000).

- اطلب من الأشخاص تقدير مهامهم، وثم قم بإضافة تقديرات المهام معًا.
- ارجع إلى الخبرة في المشاريع السابقة.
- احتفظ بالتقديرات السابقة واعرف مدى دقتها. استخدمها لضبط التقديرات الجديدة.

يتم إعطاء المؤشرات لمزيد من المعلومات حول هذه الطرق في "المصادر الإضافية لتقدير البرمجيات" في نهاية هذا القسم. هنا إليك طريقة جيدة لتقدير المشروع:

تحديد الأهداف[1]. لماذا تحتاج إلى التقدير؟ ما الذي تقدره؟ هل تقدر فقط أنشطة البناء، أو كل ما يتعلق بعملية التطوير؟ هل تقوم بتقدير فقط الجهود على مشروعك، أو مشروعك بالإضافة إلى الإجازات والعطلات والتدريب وغير ذلك من الأنشطة غير المتعلقة بالمشروع؟ ما مدى دقة التقديرات اللازمة لتحقيق أهدافك؟ ما درجة اليقين الذي يجب ربطه بالتقدير؟ هل سيؤدي تقدير متفائل أو تشاؤمي إلى نتائج مختلفة إلى حد كبير؟

اسمح بصرف وقت للتقدير، وخطط له. التقديرات السريعة هي تقديرات غير دقيقة. إذا كنت تُقدِّر مشروعًا كبيرًا، فاحرص على التعامل مع التقدير كمشروع مصغَّر وخذ الوقت الكافي لوضع خطة مصغرة للتقدير حتى تتمكن من تنفيذها جيدًا.

وضّح متطلبات البرمجيات[2]. ومثلما لا يمكن للمهندس المعماري تقدير حجم تكلفة المنزل "الكبير جدًا"، لا يمكنك تقدير مشروع برمجي "كبير جدًا" بشكل موثوق به. من غير المعقول أن يتوقع أي شخص أن تكون قادرًا على تقدير مقدار العمل المطلوب لبناء شيء عندما لم يتم تحديد هذا "الشيء". لذلك حدد المتطلبات أو التخطيط لمرحلة استكشاف أولية قبل إجراء تقدير.

قدّر على مستوى منخفض من التفاصيل. قم بإعداد التقدير على أساس فحص تفصيلي لأنشطة المشروع استنادًا إلى الأهداف التي حددتها. بشكل عام، كلما كان فحصك أكثر تفصيلًا، كلما كان تقديرك أكثر دقةً. يقول قانون الأعداد الكبيرة أن الخطأ بنسبة 10 في المئة في جزء واحد كبير سيكون 10 في المئة أعلى أو 10 في المئة أخفض. وبالنسبة لـ 50 جزء صغير، فإن بعض أخطاء الـ 10 في المئة في الأجزاء ستكون عالية وبعضها سيكون منخفضا، وسوف تميل الأخطاء إلى إلغاء بعضها البعض.

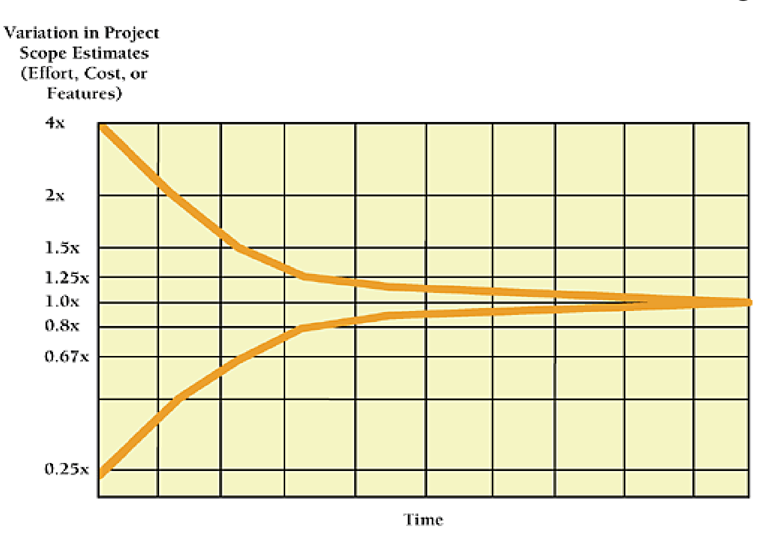
[1] **قراءة متعمقة:** هذه الطريقة مقتبسه من اقتصاد هندسة البرمجيات (بويم 1981).

[2] **إشارة مرجعية:** لمزيد من المعلومات عن متطلبات البرمجيات، انظر القسم 3 .4 "المتطلبات المسبقة".

استخدم العديد من تقنيات التقدير المختلفة، وقارن النتائج[1]. تم تحديد قائمة طرق التقدير في بداية القسم بعدة تقنيات. لن ينتج عنها النتائج نفسها، لذلك جرّب العديد منها. ومن ثم ادرس النتائج المختلفة من الطرق المختلفة. على سبيل المثال يتعلم الأطفال مبكرًا أنه إذا طلبوا من كل والد بشكل فردي الحصول على وعاء ثالث من الآيس كريم، فلديهم فرصة أفضل في الحصول على "نعم" بواحد على الأقل مما لو سألو أحد الوالدين فقط. وفي بعض الأحيان يكون الوالدان منتبهين ويعطون نفس الإجابة. في بعض الأحيان لا يفعلون ذلك. لذلك تعرّف على الإجابات المختلفة التي يمكنك الحصول عليها من أساليب تقدير مختلفة.

لا يوجد نهج هو الأفضل في جميع الظروف، ويمكن أن تكون الاختلافات فيما بينها واضحة. على سبيل المثال، بالنسبة إلى الطبعة الأولى من هذا الكتاب، كان تقديري الأساسي لمقياس طول الكتاب يتراوح بين 250 و300 صفحة. وعندما حصلت في النهاية على تقديرات متعمقة، جاءت التقديرات إلى 873 صفحة. "هذا لا يمكن أن يكون صحيح"، هكذا قلت لنفسي. لذا قدرتها مرة ثانية باستخدام تقنية مختلفة تمامًا. وجاء التقدير الثاني إلى 828 صفحة. وبالنظر إلى أن هذه التقديرات كانت في حدود خمسة بالمئة من بعضها البعض، فقد توصلت إلى أن الكتاب سيكون أقرب إلى 850 صفحة من 250 صفحة، وقد تمكنت من تعديل كتابتي للكتاب وفقًا لذلك.

قم بإعادة التقدير بشكل دوري. تتغير العوامل في مشروع برمجيات بعد التقدير الأولي، لذا خطط لتحديث تقديراتك بشكل دوري. كما يوضح الشكل 28-2، يجب أن تتحسن دقة التقديرات الخاصة بك وأنت تتحرك نحو إكمال المشروع. من وقت لآخر، قارن نتائجك الفعليّة بالنتائج المقدّرة، واستخدم هذا التقييم لتحسين التقديرات لبقية المشروع.[2]



[1] **إشارة مرجعية:** من الصعب العثور على مجال في تطوير البرامج لا يكو ... قيمة. التقدير هو إحدى الحالات التي يكون فيها التكرار مفيدًا. للحصول على ملخص للتقنيات المتكررة، انظر القسم 8.34 ... مرة بعد مرة".

[2] cc2e.com/2864

الشكل 28-2 التقديرات التي يتم إنشاؤها مبكرًا في المشروع غير دقيقة بطبيعتها. مع تقدم المشروع، يمكن أن تصبح التقديرات أكثر دقة. قُم بإعادة التقدير بشكل دوري خلال المشروع، واستخدم ما تعلمته أثناء كل نشاط لتحسين تقديرك للنشاط التالي.

قدّر كمية البناء[1]

ويعتمد المدى الذي سيكون فيه البناء ذو تأثير كبير على الجدول الزمني للمشروع بشكل جزئي على نسبة المشروع التي سيتم تخصيصها للبناء - والذي يُفهم منه التصميم التفصيلي، وكتابة الشفرة، وتصحيح الأخطاء، واختبار الوحدة. ألق نظرة أخرى على الشكل 27-3. كما يوضح الشكل، تختلف النسبة حسب حجم المشروع. وإلى أن تتوفر لدى شركتك بياناتها حول تاريخ المشروع، فإن نسبة الوقت المخصصة لكل نشاط مبيّن في الشكل هي مكان جيد لبدء التقديرات الخاصة بمشاريعك.

إن أفضل إجابة على السؤال عن مقدار البناء الذي سيطلبه المشروع هو أن النسبة ستختلف من مشروع إلى مشروع ومن منظمة إلى أخرى. احتفظ بسجلات لتجربة مؤسستك على المشاريع، واستخدمها لتقدير الوقت الذي ستستغرقه المشاريع المستقبلية.

التأثيرات على الجدول[2]

إن أكبر تأثير على الجدول الزمني لمشروع البرمجيات هو حجم البرنامج المراد إنتاجه. ولكن العديد من العوامل الأخرى تؤثر أيضًا على جدول تطوير البرمجيات. لقد حددت الدراسات الخاصة بالبرامج التجارية بعضًا من هذه العوامل، وهي موضحة في الجدول 28-1.

الجدول 28-1 العوامل التي تؤثر على جهد مشروع البرمجيات.

العامل	التأثير المحتمل المُفيد	التأثير المحتمل المُضر
الموقع المُشترك مُقابل التطوير متعدد المواقع	-14%	22%
حجم قاعدة البيانات	-10%	28%
تطابق الوثائق لاحتياجات المشروع	-19%	23%
المرونة المسموح بها في تفسير المتطلبات	-9%	10%
كيفية التعامل مع المخاطر النشطة	-12%	14%
الخبرة بالأدوات و اللغة	-16%	20%
استمرارية الأشخاص (الدوران)	-19%	29%
تقلب المنصة	-13%	30%

[1] **إشارة مرجعية:** لمزيد من التفصيل عن كمية كتابة الشفرة لمشاريع ذات أحجام مختلفة، انظر "تناسب النشاط والحجم" في القسم 27.5.

[2] **إشارة مرجعية:** لا يبدو تأثير حجم البرنامج على الإنتاجية والجودة واضحًا دائمًا بشكل بديهي. انظر إلى الفصل 27 " كيف يؤثر حجم البرنامج على عملية البناء " لتوضيح كيف يؤثر الحجم على عملية البناء.

نضج العملية	%13-	15%
تعقيد المُنتج	%27-	74%
قدرة المُبرمج	%24-	34%
الموثوقية المطلوبة	%18-	26%
متطلبات قدرة المحلل	%29-	42%
متطلبات إعادة الاستخدام	%5-	24%
التطبيقات الحديثة	%11-	12%
قيود التخزين (كم ستستهلك من سعة التخزين المتاحة)	0%	46%
تماسك الفريق	%10-	11%
خبرة الفريق في مجال التطبيقات	%19-	22%
خبرة الفريق بمنصة التكنولوجيا	%15-	19%
القيد الزمني (التطبيق نفسه)	0%	63%
استخدام أدوات البرمجيات	%22-	17%
المصدر: تقدير تكلفة البرمجيات باستخدام Cocomo II(بويم و أخرون 2000).		

فيما يلي بعض العوامل الأقل كمًا التي يمكن أن تؤثر في الجدول الزمني لتطوير البرامج. هذه العوامل مأخوذة من تقدير بويم لتكلفة البرمجيات باستخدام (Cocomo II 2000)، وطريقة كابرز جونز لتقدير تكاليف البرمجيات (1998):

- متطلبات المطور (الخبرة والقدرة)
- خبرة وقدرة المُبرمج
- حافز الفريق
- إدارة الجودة
- كمية الشفرة المُعاد استخدامها أكثر من مرة
- دوران الموظفين
- تقلبات المتطلبات
- جودة العلاقة مع العملاء
- مُشاركة المُستخدم في تحديد المتطلبات
- خبرة العميل بنوع التطبيق
- مدى مشاركة المبرمجين في تطوير المتطلبات
- بيئة الأمان المصنفة للحاسب والبرامج والبيانات
- كمية الوثائق

أهداف المشروع (الجدول الزمني مقابل الجودة مقابل القابلية للاستخدام مقابل العديد من الأهداف المحتملة الأخرى).

يمكن أن يكون كل من هذه العوامل هامًا، لذا يجب أخذها بعين الاعتبار إلى جانب العوامل الموضحة في الجدول 28-1 (والذي يتضمن بعضًا من هذه العوامل).

التقدير مُقابل التحكم[1]

يعتبر التقدير جزءًا مهمًا من التخطيط اللازم لإكمال مشروع برمجيات في الوقت المحدد. بمجرد أن يكون لديك تاريخ التسليم ومواصفات المنتج، فإن المشكلة الرئيسية هي كيفية التحكم في نفقات الموارد البشرية والمصادر التقنية لتسليم المنتج في الوقت المحدد. وبهذا المعنى، فإن دقة التقدير الأولي أقل أهمية بكثير من نجاحك اللاحق في التحكم في الموارد للوفاء بالجدول الزمني.

ماذا تفعل إذا كنت مُتخلفا عن الموعد المحدد لتسليم المشروع

يتجاوز المشروع المتوسط جدوله الزمني المخطط له بحوالي 100 بالمئة، كما ذكر في هذا الفصل. وعندما تكون متخلفا عن الوقت، فإن زيادة الوقت ليس خيارًا في العادة. إذا كان كذلك، فاتخذ هذا الخيار. بخلاف ذلك، يمكنك تجربة تطبيق واحد أو أكثر من هذه الحلول:

بيانات مثبتة

تفاءل أن تنجز في الوقت المحدد. التفاؤل المأمول هو ردة فعل شائعة على تخلف المشروع عن الجدول الزمني. عادةً ما يكون التبرير كما يلي: "استغرقت المتطلبات وقتًا أطول مما كنا نتوقعه، ولكنها أصبحت الآن متينة، لذلك نحن ملزمون بتوفير الوقت لاحقًا. وسنقوم بتعويض النقص أثناء كتابة الشفرة والاختبار." هذا بالكاد هو الحال. وقد توصلت دراسة استقصائية لأكثر من 300 مشروع برمجيات إلى أن حالات التأخير والتجاوزات تتزايد بشكل عام في نهاية المشروع (فان جنوتشن 1991). حيث لا تعوض المشاريع الوقت الضائع في وقت لاحق؛ بل أنهم يتخلفون كثيراً عن الموعد المحدد لتسليم المشروع.

وسّع الفريق. وفقًا لقانون فريد بروكس، فإن الإضافة المتأخرة لأشخاص إلى مشروع برمجيات يجعله يتأخر (بروكس 1995). إنها مثل إضافة الغاز إلى النار. إن تفسير بروكس مُقنع: فالناس الجدد يحتاجون إلى وقت لكي يتعرفوا على المشروع قبل أن يصبحوا مُنتجين. حيث يستغرق تدريبهم وقت الأشخاص الذين تم تدريبهم بالفعل. ومجرد زيادة عدد الأشخاص يزيد من تعقيد ومقدار الاتصالات داخل المشروع. يُشير بروكس إلى أن

[1] السؤال المهم هو، هل تريد التنبؤ، أم تريد السيطرة؟ - توم غيلب

حقيقة أن امرأة واحدة تستطيع أن تنجب طفلًا في تسعة أشهر لا تعني أن تسع نساء يمكن أن يلدن في شهر واحد.

لا شك في أن التحذير في قانون بروكس يجب أن يتم الاهتمام به أكثر من غيره. حيث أنه من المغري إدخال أشخاص في مشروع، ونأمل أن يحضروه في الموعد المحدد. يحتاج المديرون إلى إدراك أن تطوير البرمجيات لا يشبه تثبيت الصفائح المعدنية: فعمل المزيد من العمال لا يعني بالضرورة أنه سيتم إنجاز المزيد من العمل.

ومع ذلك، فإن العمل البسيط الذي يضيفه المبرمجون إلى مشروع متأخر يجعله لاحقًا يخفي حقيقة أنه -في بعض الظروف - من الممكن إضافة أشخاص إلى مشروع متأخر وتسريعه. وكما يشير بروكس في تحليل قانونه، فإنه لا تساعد إضافة الأشخاص إلى مشاريع برمجية لا يمكن فيها تقسيم المهام وتنفيذها بشكل مستقل. ولكن إذا كانت مهام المشروع قابلة للتقسيم، فيمكنك تقسيمها بشكل أكبر وتعيينها لأشخاص مختلفين، حتى للأشخاص الذين تمت إضافتهم متأخرين في المشروع. وقد حدد باحثون آخرون بشكل رسمي الظروف التي يمكنك من خلالها إضافة أشخاص إلى مشروع متأخر دون أن يؤدي ذلك إلى تأخره (عبد الحميد 1989، ماكونيل 1999).

قلّل من نطاق المشروع[1]. غالباً ما يتم التغاضي عن التقنية القوية لتقليل نطاق المشروع. فإذا قمت بإزالة إحدى الميزات، فإنك تُلغي تصميم وتشفير وتصحيح الأخطاء واختبار وتوثيق تلك الميزة. كما أنك تُلغي واجهة هذه الميزة لميزات أخرى.

عندما تخطط للمنتج في البداية، يمكنك تقسيم إمكانيات المنتج إلى "يجب أن يملكها"، و "من الجيد أن يمتلكها"، و "اختياريات". وإذا تأخرت عن تنفيذ هذا، فقم بتحديد أولويات "الاختيارات" و "من الجيد أن يمتلكها" وأسقط الأقل أهمية.

باختصار حول إسقاط ميزة ما، يمكنك توفير نسخة أرخص من نفس الوظيفة. وقد تقدم إصدارًا في الوقت المحدد ولكن لم يتم ضبطه على الأداء الجيد. وقد توفر إصدارًا يتم فيه تنفيذ الوظيفة الأقل أهمية بشكل سيء. قد تقرر التراجع عن متطلبات السرعة نظرًا لأنه من الأسهل جدًا تقديم نسخة بطيئة. ويمكنك التراجع عن أحد متطلبات المساحة؛ نظرًا لأنه من الأسهل توفير إصدار ذاكرة مكثفة.

---

[1] **قراءة متعمقة**: للحصول على حجة لصالح بناء الميزات الأكثر حاجة فقط، انظر الفصل 14 " التحكم في مجموعة الميزات" في التطوير السريع (ماكونيل 1996).

أعد تقدير وقت التطوير للميزات الأقل أهمية. ما الوظائف التي يمكنك توفيرها خلال ساعتين أو يومين أو أسبوعين؟ ما الذي تكتسبه عن طريق بناء الإصدار الذي يستغرق أسبوعين بدلًا من إصدار يومين، أو إصدار يومين بدلاً من الإصدار الذي يستغرق ساعتين؟

مصادر إضافية عن تقدير البرمجيات[1]

فيما يلي بعض المراجع الإضافية حول تقدير البرمجيات:

Boehm, Barry, et al. Software Cost Estimation with Cocomo II. Boston, MA: Addison-Wesley, 2000.

بويم، باري، وآخرون. تقدير تكلفة البرمجيات باستخدام كوكومو 2، أديسون ويسلي، 2000. يصف هذا الكتاب خصوصيات وعموميات نموذج التقدير Cocomo II، والذي يعتبر بلا شك النموذج الأكثر شعبية في الاستخدام اليوم.

Boehm, Barry W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice Hall, 1981.

بويم، باري. اقتصاد هندسة البرمجيات. إنجلوود كليفس، نيوجيرسي: برنتيس هول، 1981. يحتوي هذا الكتاب الأقدم على معالجة شاملة لتقدير مشروعات البرمجيات التي يُنظر إليها بشكل أعم من كتاب بويم الجديد.

Humphrey, Watts S. A Discipline for Software Engineering. Reading, MA: Addison-Wesley, 1995.

همفري، واتس. الانضباط في هندسة البرمجيات. ريدينغ، أديسون ويسلي، 1995. يصف الفصل الخامس من هذا الكتاب طريقة التحقق لهمفري، وهي تقنية لتقدير العمل على مستوى المطور الفردي.

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. Software Engineering Metrics and Models. Menlo Park, CA: Benjamin/Cummings, 1986.

كونتي، دونسمور، و شين. نماذج ومقاييس هندسة البرمجيات. مينلو بارك، بنيامين / كامينغز، 1986. يحتوي الفصل 6 على مسح جيد لتقنيات التقدير، بما في ذلك تاريخ التقديرات، والنماذج الإحصائية، والنماذج النظرية، والنماذج المركبة. يوضح الكتاب أيضًا استخدام كل أسلوب تقدير في قاعدة بيانات المشاريع، ويقارن التقديرات بالأطوال الفعلية للمشروع.

Gilb, Tom. Principles of Software Engineering Management. Wokingham, England: Addison-Wesley, 1988.

[1] cc2e.com/2871

جيلب، توم. مبادئ إدارة هندسة البرمجيات. ووكينغهام، إنكلترا، أديسون ويسلي، 1988. عنوان الفصل 16، "المبادئ العشر لتقدير سمات البرمجيات"، إنه ساخر نوعاً ما. يجادل جيلب ضد تقدير المشروع لصالح مراقبة المشروع. بالإشارة إلى أن الأشخاص لا يرغبون حقًا في التنبؤ بدقة ولكنهم يرغبون في التحكم في النتائج النهائية، يضع جيلب 10 مبادئ يمكنك استخدامها لتوجيه مشروع لالتزام الموعد النهائي للتسليم، أو هدف التكلفة، أو هدف مشروع آخر.

4.28 القياس Measurement

يمكن قياس مشاريع البرمجيات بعدة طرق. فيما يلي سببان قويان لقياس العملية:

نقطة مفتاحية

بالنسبة لأي سمة مشروع، من الممكن أن يفوق قياس هذه السمة عدم قياسها على الإطلاق. قد لا يكون القياس دقيقًا تمامًا، وقد يكون من الصعب إجراؤه، وقد تحتاج إلى تحسينه بمرور الوقت، ولكن القياس سيعطيك طريقة للتعامل مع عملية تطوير البرامج التي لا تملكها بدونها (جيلب 2004).

وإذا كانت هذه البيانات ستستخدم في تجربة علمية، فيجب تحديدها كميًا. هل يمكنك أن تتخيل عالمًا يوصي بفرض حظر على منتج غذائي جديد لأن مجموعة من الفئران البيضاء "يبدو أنها أكثر مرضًا" من مجموعة أخرى؟ هذا سخيف. أنت تطالب بسبب كمي، مثل "الجرذان التي تناولت المنتج الغذائي الجديد كانت مريضة 3.7 يومًا في الشهر أكثر من الجرذان التي لم تفعل ذلك." لتقييم أساليب تطوير البرمجيات، يجب قياسها. تصريحات مثل "هذه الطريقة الجديدة تبدو أكثر إنتاجية" ليست جيدة بما فيه الكفاية.

احذر من الآثار الجانبية للقياس[1]. القياس له تأثير تحفيزي. حيث يلتفت انتباه الناس إلى أي شيء يتم قياسه، بافتراض استخدام هذا القياس لتقيمهم. اختر ما تقيسه بعناية. حيث يميل الناس إلى التركيز على العمل الذي يتم قياسه وتجاهل العمل الذي لا يتم قياسه.

أن تجادل ضد القياس، يعني أن تجادل أنه من الأفضل عدم معرفة ما يحدث بالفعل في مشروعك. عندما تقيس جانبًا من مشروع ما، فإنك ستعرف شيئًا لم تعرفه من قبل. ويمكنك معرفة ما إذا كان هذا الجانب يكبر أو يصغر أو يبقى على حاله. ويمنحك القياس نافذة على الأقل على هذا الجانب من مشروعك. قد تكون النافذة صغيرة وداكنة حتى تقوم بتحسين قياساتك، ولكنها ستكون أفضل من عدم وجود نافذة على الإطلاق. لذلك للمجادلة ضد جميع القياسات لأن بعضها غير حاسم هي المجادلة ضد النوافذ لأن البعض من الممكن أن يكون غائمًا.

يمكنك عمليًا قياس أي جانب من جوانب عملية تطوير البرمجيات. يسرد الجدول 28-2 بعض القياسات التي وجدها الممارسون الآخرون مفيدة.

[1] ما يمكن قياسه يمكن القيام به. — توم بيترز

جدول 2-28 قياسات مفيدة لعملية تطوير البرمجيات

الحجم
مجموع أسطر الشفرة المكتوبة
مجموع أسطر التعليقات
مجموع الصفوف و الإجرائيات
مجموع تصريحات المعطيات
مجموع الأسطر الفارغة
تتبع العيوب
شدة كل عيب
موقع كل عيب (صف أو إجرائية)
أصل كل عيب (المتطلبات، التصميم، البناء، الاختبار)
الطريقة التي يتم بها تصحيح كل عيب
الشخص المسؤول عن كل عيب
عدد السطور المتأثرة بكل تصحيح لعيب
ساعات العمل المقضية في تصحيح كل عيب
متوسط الوقت اللازم للعثور على عيب
متوسط الوقت اللازم لإصلاح عيب
عدد المحاولات المبذولة لتصحيح كل عيب
عدد الأخطاء الجديدة الناتجة عن تصحيح العيوب
الإنتاجية
عدد ساعات العمل المصروفة على المشروع
ساعات العمل المصروفة على كل صف أو إجرائية
عدد المرات التي يتغير فيها كل صف أو إجرائية
كمية الدولارات المصروفة على المشروع
كمية الدولارات المصروفة على كل سطر من الشفرة
كمية الدولارات المصروفة على كل عيب

يمكنك جمع معظم هذه القياسات باستخدام أدوات البرمجيات المتوفرة حاليًا. تشير المناقشات في جميع أنحاء الكتاب إلى الأسباب التي تجعل كل قياس مفيدًا. في هذا الوقت، فإن معظم القياسات ليست مفيدة في التمييز الدقيق بين البرامج، والصفوف، والإجرائيات (شيبرد و إنس 1989).

إنها مفيدة بشكل أساسي لتحديد الإجرائيات "المتطرفة"؛ تعتبر القياسات غير الطبيعية في الإجرائية علامة تحذير مفادها أنه يجب عليك إعادة فحص هذه الإجرائية، والتحقق من الجودة المنخفضة غير العادية.

إذا بدأت من خلال جمع البيانات حول جميع القياسات الممكنة — ستغرق نفسك في بيانات معقدة للغاية بحيث لن تتمكن من معرفة ما يعنيه أيًا منها. بل ابدأ بمجموعة بسيطة من القياسات، مثل عدد العيوب، وعدد أشهر

العمل، ومجموع الدولارات المصروفة، وإجمالي عدد أسطر الشفرة. وقم بتوحيد المقاييس عبر مشاريعك، ثم قم بتنقيتها والإضافة إليها عندما يتحسن فهمك لما تريد قياسه (بيتراسانتا 1990).

تأكد من أنك تجمع البيانات لسبب ما. حدد الأهداف، وحدد الأسئلة التي تحتاج إلى طرحها لتحقيق الأهداف، ثم قم بقياس الإجابة على الأسئلة (باسيلى وويس 1984). تأكد من أنك لا تطلب سوى أكبر قدر ممكن من المعلومات التي يمكن الحصول عليها، وتذكر أن جمع البيانات سيظل دائمًا في المقعد الخلفي إلى المواعيد النهائية (باسيلي وآخرون 2002).

مصادر إضافية عن إدارة البرمجيات[1]

فيما يلي المصادر الإضافية:

Oman, Paul and Shari Lawrence Pfleeger, eds. Applying Software Metrics. Los Alamitos, CA: IEEE Computer Society Press, 1996.

عُمان وبول وشاري لورنس بفليجر، محرران. تطبيق قياسات البرمجيات. لوس الأليموس، كاليفورنيا: جمعية الصحافة الحاسوب IEEE، 1996. يجمع هذا المجلد أكثر من 25 ورقة أساسية حول قياس البرمجيات تحت غطاء واحد.

Jones, Capers. Applied Software Measurement: Assuring Productivity and Quality, 2d ed. New York, NY: McGraw-Hill, 1997.

جونز، كابير. قياس البرمجيات التطبيقية: ضمان الإنتاجية والجودة، الإصدار الثاني. نيويورك: مكجراو هيل، 1997. جونز هو رائد في قياس البرمجيات، وكتابه هو تراكم معرفي في هذا المجال. ويوفر النظرية النهائية والممارسة العملية لتقنيات القياس الحالية ويصف المشاكل مع القياسات التقليدية. ويضع برنامجًا كاملاً لجمع "مقاييس الوظيفة". جمع جونز وحلل كمية هائلة من بيانات الجودة والإنتاجية ويستخلص هذا الكتاب النتائج في مكان واحد، بما في ذلك فصل رائع عن الأرقام المتوسطة لتطوير البرمجيات في الولايات المتحدة.

Grady, Robert B. Practical Software Metrics for Project Management and Process Improve-ment. Englewood Cliffs, NJ: Prentice Hall PTR, 1992.

جرادي، روبرت ب. مقاييس البرمجيات العملية لإدارة المشاريع وتحسين العمليات. إنجلوود كليفس، نيو جيرسي: برنتيس هول PTR، 1992. يشرح جرادي الدروس المستفادة من إنشاء برنامج لقياس البرمجيات في هيوليت باكارد، ويخبرك بكيفية إنشاء برنامج لقياس البرمجيات في مؤسستك.

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. Software Engineering Metrics and Models. Menlo Park, CA: Benjamin/Cummings, 1986.

[1] cc2e.com/2878

كونتي، و دونسمور، وشين. نماذج ومقاييس هندسة البرمجيات. متنزه مينلو، كريس: بينجامين / كامينغز، 1986. يبرز هذا الكتاب المعرفة الحالية لقياس البرمجيات حوالي عام 1986، بما في ذلك القياسات المستخدمة بشكل شائع، والتقنيات التجريبية، ومعايير تقييم النتائج التجريبية.

Basili, Victor R., et al. 2002. "Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory," Proceedings of the 24th International Conference on Software Engineering. Orlando, FL, 2002.

باسيلي، وفيكتور، وآخرون. 2002. "الدروس المستفادة من 25 سنة من تحسين العملية: صعود وهبوط مختبر هندسة البرمجيات في ناسا". وقائع المؤتمر الدولي الرابع والعشرين لهندسة البرمجيات. اورلاندو، فلوريدا، 2002. تبرز هذه الورقة الدروس المستفادة من واحدة من أكثر منظمات تطوير البرمجيات تطورًا في العالم. حيث تُركز الدروس على موضوعات القياس.

NASA Software Engineering Laboratory. Software Measurement Guidebook, June 1995, NASA-GB-001-94.

مختبر هندسة البرمجيات في ناسا. دليل قياس البرمجيات، يونيو 1995، NASA-GB-001-94. ربما يكون هذا الدليل الذي يضم حوالي 100 صفحة هو أفضل مصدر للمعلومات العملية حول كيفية إعداد برنامج قياس وتشغيله. ويمكن تنزيله من موقع NASA الإلكتروني.

Gilb, Tom. Competitive Engineering. Boston, MA: Addison-Wesley, 2004.

غليب، وتوم. الهندسة التنافسية. بوسطن، أديسون ويسلي، 2004. يقدم هذا الكتاب مقاربة تركز على القياس لتحديد المتطلبات، وتقييم التصاميم، وقياس الجودة، وإدارة المشاريع بشكل عام. يمكن تنزيله من موقع غليب الإلكتروني.

5.28 التعامل مع المبرمجين كأشخاص

يُطلب التجريد للنشاط البرمجي كموازنة طبيعية في بيئة المكتب والاتصالات الكثيرة بين زملاء العمل. حيث تقدم شركات التقنية العالية المنتزهات كحرم الشركة، والبنى التنظيمية الأساسية، والمكاتب المريحة، وغيرها من الميزات البيئية "عالية المحسوسية" لموازنة الانفعال الحاد، وأحيانًا العقلية العقيمة للعمل في حد ذاته.

تدمج أكثر الشركات التقنية نجاحًا بين عناصر التكنولوجيا المتقدمة والملموسية العالية (نايسبت 1982). يصف هذا القسم الطرق التي يكون فيها المبرمجون أكثر من انعكاسات عضوية لشخصياتهم الثانية السيليكونية.

كيف يقضي المبرمجون وقتهم؟

يقضي المبرمجون وقتهم في البرمجة، ولكنهم أيضًا يقضون وقتًا في الاجتماعات، والتدريب، وقراءة بريدهم، وفي التفكير فقط. وجدت دراسة أجريت عام 1964 في مختبرات بيل أن المبرمجين أمضوا وقتهم بهذه الطريقة كما هو موضح في الجدول 3-28.

الجدول 3-28 عرض واحد لكيفية قضاء المبرمجين وقتهم

النشاط	شفرة المصدر	العمل	الشخصي	المقابلات	التدريب	البريد / متفرقات مستندات	كتيبات تقنية	إجراءات التشغيل، متفرقات	اختبار البرنامج	المجموع
التكلم أو الإصغاء	4%	17%	7%	3%				1%		32%
التكلم مع المدير		1%								1%
التلفون		2%	1%							3%
القراءة	14%					2%	2%			18%
الكتابة/ التسجيل	13%					1%				14%
بعيد أو خارج		4%	1%	4%	6%					15%
المشي	2%	2%	1%			1%				6%
متنوع	2%	3%	3%			1%		1%	1%	11%
المجموع	35%	29%	13%	7%	6%	5%	2%	2%	1%	100%

المصدر: "الدراسات البحثية للمبرمجين والبرمجة" (بايردين 1964، ذكرت في بويم 1981).

تستند هذه البيانات على دراسة الوقت والحركة لـ 70 مبرمجًا. البيانات قديمة، ونسب الوقت المقضي في الأنشطة المختلفة تختلف بين المبرمجين، ولكن هذه النتائج مع ذلك تحفز على التفكير.

يتم صرف حوالي 30 بالمئة من وقت المبرمج في الأنشطة غير الفنية التي لا تساعد المشروع بشكل مُباشر: المشي والأعمال الشخصية وما إلى ذلك. قضى المبرمجون في هذه الدراسة ستة في المئة من وقتهم في المشي؛ هذا حوالي 2.5 ساعة في الأسبوع، حوالي 125 ساعة في السنة. قد لا يبدو ذلك كثيرًا حتى تدرك أن المبرمجين يقضون من الوقت كل عام على المشي نفس الكمية من الوقت في التدريب، وثلاثة أضعاف الوقت الذي يقضونه في قراءة الكتيبات التقنية، وستة أضعاف ما يقضونه في التحدث مع مديريهم. أنا شخصيا لم أر الكثير من التغيير في هذا النمط في صرف الوقت في هذه الأيام.

تباين بين الأداء والجودة

بيانات مثبتة

تختلف الموهبة والجهد بين المبرمجين الفرديين بشكل كبير، كما هما في جميع المجالات. وجدت إحدى الدراسات أنه في مجموعة متنوعة من المهن– الكتابة، كرة القدم، الابتكار، عمل الشرطة،

وتجريب الطائرات - أنتجت أعلى 20 في المئة من الناس حوالي 50 في المئة من الناتج (أوغسطين 1979). تستند نتائج الدراسة إلى تحليل بيانات الإنتاجية، مثل الأهداف، وبراءات الاختراع، والحالات التي تم حلها، وما إلى ذلك. نظرًا لأن بعض الأشخاص لا يقدمون أي مساهمة ملموسة على الإطلاق، ولم يتم أخذهم بالاعتبار في الدراسة (ذوي الدخل المتوسط الذين لا يتعاملون مع أي شيء ملموس، أو مخترعين لا يمتلكون براءات اختراع، أو محققين لم يقوموا بإغلاق قضايا، وما إلى ذلك)، فمن المحتمل أن هذه البيانات تقلل من الاختلاف الفعلي في الإنتاجية.

في البرمجة على وجه التحديد أظهرت العديد من الدراسات اختلافات ترتيب المقدار في جودة البرامج المكتوبة، وأحجام البرامج المكتوبة، وإنتاجية المبرمجين.

التباين الفردي

بيانات مثبتة

أجريت الدراسة الأصلية التي أظهرت اختلافات كبيرة في إنتاجية البرامج الفردية في أواخر الستينات من قبل ساكمان و اريكسون و كرانت (1968). حيث درسوا المبرمجين المحترفين بمتوسط خبرة 7 سنوات، ووجدوا أن نسبة وقت كتابة الشفرة المبدئي بين أفضل وأسوأ المبرمجين كانت حوالي 20 إلى 1، ونسبة مرات تصحيح الأخطاء أكثر من 25 إلى 1، ونسبة حجم البرنامج 5 إلى 1، ونسبة سرعة تنفيذ البرنامج حوالي 10 إلى 1. ولكنهم لم يعثروا على أي علاقة بين مقدار خبرة المبرمج وجودة الشفرة أو الإنتاجية.

بيانات مثبتة

على الرغم من أن النسب المحددة مثل 25 إلى 1 ليست ذات مغزى بشكل خاص، ولكن العبارات الأكثر عمومية مثل "هناك اختلافات في ترتيب الأهمية بين المبرمجين" ذات مغزى وقد تم تأكيدها بواسطة العديد من الدراسات الأخرى للمبرمجين المحترفين (كورتيس 1981، ميلز 1983، ديماركو وليستر 1985، كورتيس وآخرون 1986، كارد 1987، بويم وباباتشيو 1988، فالت ومكراري 1989، بويم وآخرون 2000).

تباين الفريق

تُظهر فرق البرمجة أيضًا اختلافات كبيرة في جودة البرامج والإنتاجية. يميل المبرمجون الجيدون إلى التجمع، كما يفعل المبرمجون السيئون، وهي ملاحظة أكدتها دراسة أجريت على 166 مبرمجًا محترفًا من 18 منظمة (ديماركو وليستر 1999).

بيانات مثبتة

في دراسة واحدة من سبعة مشاريع متطابقة، تنوعت الجهود المبذولة من عامل 3.4 إلى 1 وحجم البرنامج بعامل 3 إلى 1 (بويم و غراي و سيوالدت 1984). على الرغم من مدى الإنتاجية، لم يكن المبرمجون في هذه الدراسة مجموعة متنوعة. كانوا جميعًا مبرمجين محترفين لديهم عدة سنوات من الخبرة

ممن التحقوا ببرنامج الدراسات العليا في علوم الحاسوب. فمن المعقول أن نفترض أن دراسة مجموعة أقل تجانسًا من المبرمجين ستُظهر اختلافات أكبر.

لاحظت دراسة سابقة لفرق البرمجة وجود فرق 5 إلى 1 في حجم البرنامج وتغير من 2.6 إلى 1 في الوقت اللازم للفريق لاستكمال نفس المشروع (واينبرغ وشولمان 1974).

بيانات مثبتة

بعد مراجعة أكثر من 20 عامًا من البيانات حول بناء نموذج تقدير كوكومو Cocomo II 2، خلص باري بويهم وغيره من الباحثين إلى أن تطوير برنامج مع فريق في المرتبة 15 من 100 للمبرمجين حسب القدرة يتطلب عادة حوالي 3.5 ضعف من أشهر العمل لتطوير برنامج مع فريق في المرتبة 90 من 100 (بويم وآخرون 2000). وقد وجد بويم وغيره من الباحثين أن 80 بالمئة من المساهمة تأتي من 20 بالمئة من المساهمين (بويم b1987).

حيث الآثار المترتبة على التطويع والتوظيف واضحة. إذا كان عليك دفع المزيد للحصول على مبرمج ترتيبه أعلى من 10 بالمئة بدلًا من مبرمج ترتيبه أقل من 10 بالمئة قم باستغلال الفرصة. عندها ستحصل على مكافأة فورية لجودة وإنتاجية المبرمج الذي توظفه، وستحصل على أثر متبقي في جودة وإنتاجية المبرمجين الآخرين الذين تستطيع مؤسستك الاحتفاظ بهم لأن المبرمجين الجيدين يميلون إلى التجمع مع بعضهم البعض.

قضايا مذهبية

لا يدرك مدراء مشاريع البرمجة دائمًا أن بعض مشكلات البرمجة هي أمور تتعلق بالدين. إذا كنت مديرًا وحاولت مطالبًا بالامتثال لممارسات برمجة معينة، فأنت توجه الدعوة إلى غضب المبرمجين. إليكم فيما يلي قائمة من القضايا الدينية:

- لغة البرمجية
- أسلوب المسافة البادئة
- وضع الأقواس
- خيار بيئة التطوير المتكاملة IDE
- أسلوب كتابة التعليقات
- المقايضة بين الكفاءة وقابلية القراءة
- خيار المنهجية– على سبيل المثال سكرم Scrum أو البرمجة المتطرفة أو التوزيع التطوري.
- المرافق البرمجية
- اصطلاحات التسميّة
- استخدام تعليمات goto
- استخدام المتغيرات العامة

القياسات، خاصة مقاييس الإنتاجية مثل عدد سطور الشفرة في اليوم

القاسم المشترك بين هذه المواضيع هو أن وضع المبرمج مع كل منها هو انعكاس لأسلوبه الشخصي. إذا كنت تعتقد أنك بحاجة إلى التحكم في مبرمج في أي من هذه المناطق الدينية، فخذ بعين الاعتبار النقاط التالية:

كُن حذرًا وأنت تتعامل مع منطقة حساسة. اسأل المبرمج عن رأيه في كل موضوع حساس، قبل الدخول به بكلتا قدميك.

استخدم كلمة "اقتراحات" أو "إرشادات" فيما يتعلق بهذه المنطقة الحساسة. تجنّب وضع "قواعد" صارمة أو "معايير".

البراعة في القضايا التي يمكنك القيام بها عن طريق تجنب التفويضات الصريحة. للحصول على البراعة في نمط المسافة البادئة أو مكان وضع الأقواس، يجب تشغيل شفرة المصدر من خلال طابعة جميلة قبل أن يتم الإعلان عن الشفرة منتهية. دع الطابعة الجميلة تقوم بالتنسيق. وللدقة في أسلوب التعليق، يجب مراجعة جميع التعليمات البرمجية في الشفرة وتعديل غير الواضح منها إلى أن تصبح واضحة.

اجعل مبرمجيك يطورون معاييرهم الخاصة بهم. كما ذكر في مكان آخر، غالبًا ما تكون تفاصيل معيار محدد أقل أهمية من حقيقة وجود بعض المعايير. لا تضع معايير لمبرمجيك، ولكن أصر على توحيدهم لمعيار محدد في المجالات المهمة بالنسبة لك.

أي من المواضيع المذهبية مهمة بما فيه الكفاية لتبرير الذهاب إلى الجدل إن التناغم في الأمور الثانوية للأسلوب في أي مجال قد لا يحقق فائدة كافية لتعويض تأثيرات انخفاض الروح المعنوية. إذا وجدت الاستخدام العشوائي لـ gotos أو المتغيرات العامة، أو الأنماط غير القابلة للقراءة، أو الممارسات الأخرى التي تؤثر على المشاريع بأكملها، فكن مستعدًا للتعامل مع بعض الخلافات مع المبرمجين لتحسين جودة الشفرة. وإذا كان مبرمجوك ذوو ضمير حي، فنادرًا ما تكون هذه مشكلة. تميل أكبر المعارك إلى أن تكون على الفروق الدقيقة في أسلوب كتابة الشفرة، ويمكنك البقاء بعيدًا عن أولئك الذين لا يشكلون خسارة للمشروع.

البيئة الفيزيائية

إليك تجربة: اذهب إلى الريف، وابحث عن مزرعة، واعثر على أحد المزارعين، واسأل عن مقدار الأموال في المعدات التي يوفرها المزارع لكل عامل. سوف ينظر المزارع إلى الحظيرة ويرى بعض الجرارات، وبعض العربات، وحصادة القمح، وحصادة البازلاء، وسوف يخبرك أن المجموع يزيد عن 100000 دولار لكل عامل.

ومن ثم اذهب إلى المدينة، وابحث عن محل برمجي، واعثر على مدير برمجي، واسأل عن مقدار الأموال في المعدات التي يوفرها المدير البرمجي لكل عامل. سيطلع المدير البرمجي على مكتب ويرى مكتب وكرسي وعدد قليل من الكتب وجهاز حاسوب وسيخبرك أن المجموع أقل من 25000 دولار لكل عامل.

تُحدث البيئة الفيزيائية فرقًا كبيرًا في الإنتاجية. سأل ديماركو وليستر 166 مبرمجًا من 35 مؤسسة حول جودة بيئاتهم الفيزيائية. صنف معظم الموظفين أماكن عملهم بأنها غير مقبولة. وفي منافسة برمجية لاحقة، كان لدى المبرمجين الذين أدوا أعلى 25 في المئة مكاتب أكبر وأكثر هدوءً وأكثر خصوصية وإعاقات أقل من الناس والمكالمات الهاتفية. فيما يلي ملخص للاختلافات في المساحات المكتبية بين الأفضل والأسوأ أداءً:

العامل البيئي	أعلى 25٪	أخفض 25٪
المساحة المخصصة	78 قدم مربع	46 قدم مربع
مساحة عمل هادئة مقبولة	57% نعم	29% نعم
مساحة عمل منعزلة مقبولة	62% نعم	19% نعم
القدرة على وضع الهاتف في الوضع الصامت	52% نعم	10% نعم
القدرة على تحويل المكالمات	76% نعم	19% نعم
المقاطعات المتكررة التي لا داعي لها	38% نعم	76% نعم
مساحة العمل التي تجعل المبرمج يشعر بالتقدير	57% نعم	29% نعم
المصدر: بيوبليور (ماركو وليستير 1999).		

توضح البيانات وجود علاقة قوية بين الإنتاجية ونوعية مكان العمل. فالمبرمجون في أعلى 25 في المئة هم أكثر إنتاجية بمقدار 2.6 مرة من المبرمجين في أسفل 25 في المئة. يعتقد كل من ماركو و ليستر قد يكون لدى المبرمجين الأفضل مكاتب أفضل بشكل طبيعي لأنه تمت ترقيتهم، لكن المزيد من الفحص أظهر أن الأمر ليس كذلك. حيث كان لدى المبرمجون من نفس المنظمات تسهيلات مماثلة، بغض النظر عن الاختلافات في أدائهم.

ولقد واجهت مؤسسات البرمجة الكبيرة الكثير من التجارب المماثلة. أشارت كل من الشركات إكسيروس Xerox وتي أر دبل يو TRW وأي بي إم IBM وبيل لابس Bell Labs إلى أنها تحقق إنتاجية محسنة بشكل كبير من خلال استثمار رأسمالي يتراوح بين 10000 و 30000 دولار لكل شخص، وهي مبالغ كانت أكبر من ما تم استرداده في الإنتاجية المحسنة (بويم 1987a). مع "مكاتب الإنتاجية"، تراوحت تقديرات التقارير الذاتية من 39 إلى 47 بالمئة من التحسن في الإنتاجية (بويم وأخرون 1984).

باختصار، إذا كان مكان عملك هو بيئة أقل 25 في المئة، فيمكنك تحقيق تحسن في الإنتاجية بنسبة 100 في المئة من خلال جعله بيئة أعلى 25 بالمئة. إذا كان مكان عملك متوسطًا، فلا يزال بإمكانك تحقيق تحسن في الإنتاجية بنسبة 40 بالمئة أو أكثر بجعله بيئة أعلى 25 بالمئة.

مصادر إضافية عن المبرمجين كبشر [1]

فيما يلي مصدر إضافية:

Weinberg, Gerald M. The Psychology of Computer Programming, 2d ed. New York, NY: Van Nostrand Reinhold, 1998.

واينبرغ، جيرالد. علم نفس برمجة الحاسوب، الإصدار الثاني. نيويورك: فان نوستراند رينهولد، 1998. هذا هو الكتاب الأول الذي يحدد بوضوح المبرمجين كبشر، وما زال الأفضل عن البرمجة كنشاط بشري. انه مكتظ بملاحظات عميقة حول الطبيعة البشرية للمبرمجين وآثارها.

DeMarco, Tom and Timothy Lister. Peopleware: Productive Projects and Teams, 2d ed. New York, NY: Dorset House, 1999.

ديماركو، توم، تيموثي ليستر. المشاريع الإنتاجية وفرق العمل، الإصدار الثاني. نيويورك، نيويورك: دورست هاوس، 1999. كما يوحي العنوان، يتناول هذا الكتاب أيضًا العامل البشري في معادلة البرمجة. إنه مليء بالحكايات حول إدارة الناس، وبيئة المكتب، وتوظيف وتطوير الأشخاص المناسبين، والفرق المتنامية، والاستمتاع بالعمل. يرتكز المؤلفون على الحكايات لدعم بعض وجهات النظر غير المألوفة ومنطقهم ضعيف، لكن ما هو مهم هو روح الكتاب المتمحورة حول الناس، ويقدم المؤلفون هذه الرسالة دون تعثر.

McCue, Gerald M. “IBM’s Santa Teresa Laboratory—Architectural Design for Program Development,” IBM Systems Journal 17, no. 1 (1978): 4–25.

مكو، جيرالد. "مختبر سانتا تيريزا لشركة IBM - التصميم الهيكلي لتطوير البرامج"، مجلة أنظمة IBM 17، رقم 1(1978):4-25. يصف مكو العملية التي استخدمتها أي بي إم IBM لإنشاء مجمع مكاتبها في سانتا تيريزا. درست شركة IBM احتياجات المبرمج، ووضعت إرشادات هيكلية، وصممت المرافق مع وضع المبرمجين في الاعتبار. وشارك المبرمجون طوال الوقت. والنتيجة هي أنه في استطلاعات الرأي السنوية في كل عام، يتم تصنيف المرافق المادية في منشأة سانتا تيريزا على أعلى مستوى في الشركة.

McConnell, Steve. Professional Software Development. [2]Boston, MA: Addison-Wesley, 2004.

ماكونيل، ستيف. تطوير البرمجيات المحترفة. بوسطن، أديسون ويسلي، 2004. يلخص الفصل 7، "الأيتام المفضلين"، دراسات حول التركيبة السكانية للمبرمجين، بما في ذلك أنواع الشخصيات والخلفيات التعليمية وآفاق العمل.

[1] cc2e.com/2806

[2] cc2e.com/2820

Carnegie, Dale. How to Win Friends and Influence People, Revised Edition. New York, NY: Pocket Books, 1981.

كارنيجي، دايل. كيف تكسب الأصدقاء وتؤثر في الناس، طبعة منقحة. نيويورك: 1981. عندما كتب ديل كارنيجي عنوان الطبعة الأولى من هذا الكتاب في عام 1936، لم يكن بإمكانه إدراك الدلالة التي يحملها اليوم. يبدو مثل كتاب ميكيافيلي الذي كان على رفه. إن روح الكتاب تعارض بشكل تام تلاعب ميكافيلي، وأحد النقاط الرئيسية في كتاب كارنيجي هو أهمية تطوير الاهتمام الحقيقي بالأشخاص الآخرين. لدى كارنيجي نظرة ثاقبة في العلاقات اليومية ويشرح كيفية العمل مع الآخرين من خلال فهمهم بشكل أفضل. يمتلئ الكتاب بحكايات لا تُنسى، أحيانًا ما بين اثنين أو ثلاثة صفحات. يجب على أي شخص يعمل مع الناس قراءته في مرحلة ما، وعلى أي شخص يدير الناس قراءته الآن.

6.28 إدارة مديرك

في تطوير البرمجيات، المدراء غير التقنيين شائعون، وكذلك المدراء الذين لديهم خبرة تقنية ولكنهم متأخرين 10 سنوات. من النادر تقنيًا، وجود مديرين تقنيين حاليين. إذا كنت تعمل مع واحد منهم، افعل ما بوسعك للحفاظ على وظيفتك. انه علاج غير عادي.

إذا كان مديرك أكثر نموذجية، فأنت تواجه مهمة لا تحسد عليها لإدارة مديرك[1]. "إدارة مديرك" تعني أنك بحاجة إلى إخبار مديرك بما يجب القيام به بدلًا من العكس. الخدعة هي القيام بذلك بطريقة تسمح لمديرك بالاستمرار في الاعتقاد بأنك الشخص الذي تتم إدارته. فيما يلي بعض الطرق للتعامل مع مديرك:

- قم بإعداد أفكار حول ما تريد القيام به، ثم انتظر حتى يقوم المدير الخاص بك بإجراء عصف ذهني (فكرتك) حول القيام بما تريد القيام به.
- علّم مديرك الطريقة الصحيحة لفعل الأشياء. هذه وظيفة مستمرة لأن المدراء يتم ترقيتهم أو نقلهم أو فصلهم.
- ركّز على اهتمامات مديرك، وافعل ما يريد منك فعله حقًا، ولا تشغل مديرك بتفاصيل التنفيذ غير الضرورية. (فكر في الأمر على أنه "تغليف" لعملك).
- ارفض القيام بما يخبرك به مديرك، وأصر على القيام بواجبك بالطريقة الصحيحة.
- اعثر على وظيفة أخرى.

أفضل حل على المدى الطويل هو محاولة تثقيف مديرك. هذه ليست مهمة سهلة دائمًا، ولكن إحدى الطرق التي يمكنك الاستعداد لها هي قراءة كتاب ديل كارنيجي "كيف تكسب الأصدقاء وتؤثر في الناس".

[1] في التسلسل الهرمي، يميل كل موظف إلى الارتقاء إلى مستوى عدم كفائته- مبدأ بيتر.

مصادر إضافية عن إدارة عملية البناء[1]

فيما يلي بعض الكتب التي تتناول المشكلات ذات الاهتمام العام في إدارة مشروعات البرمجيات:

Gilb, Tom. Principles of Software Engineering Management. Wokingham, England: Addison-Wesley, 1988.

جيلب، وتوم. مبادئ إدارة هندسة البرمجيات. ووكينغهام، إنكلترا، أديسون ويسلي 1988. رسم جيلب مساره الخاص لمدة ثلاثين عامًا، وفي معظم الأوقات كان متقدمًا على مجموعته ما إذا كانت المجموعة تدرك ذلك أم لا. وهذا الكتاب مثال جيد. كان هذا الكتاب واحدًا من أوائل الكتب التي ناقشت ممارسات نمو التطوير، وإدارة المخاطر، واستخدام عمليات الفحص الرسمية. إن جيلب على وعي تام بالمناهج الرائدة؛ في الواقع، يحتوي هذا الكتاب الذي نشر منذ أكثر من 15 عامًا على معظم الممارسات الجيدة التي تُحلق حاليًا تحت شعار "رشيقة". جيلب هو إنسان عملي بشكل لا يصدق والكتاب لا يزال واحدًا من أفضل كتب إدارة البرمجيات.

McConnell, Steve. Rapid Development. Redmond, WA: Microsoft Press, 1996.

ماكونيل، ستيف. التطور السريع. ريدموند، واشنطن: ميكروسوفت برس، 1996. يغطي هذا الكتاب قضايا قيادة المشاريع وإدارة المشاريع من منظور المشاريع التي تعاني من ضغط جدول زمني كبير، والتي هي من واقع خبرتي في معظم المشاريع.

Brooks, Frederick P., Jr. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2d ed). Reading, MA: Addison-Wesley, 1995.

بروكس، فريدريك، مقالات في هندسة البرمجيات، طبعة الذكرى (الإصدار الثاني). رديدينغ: أديسون ويسلي، 1995. هذا الكتاب هو خليط من الاستعارات والفولكلور المتعلق بإدارة مشاريع البرمجة. إنه مسلي، وسيمنحك العديد من الأفكار الواضحة في مشاريعك الخاصة. يعتمد ذلك على تحديات بروكس في تطوير نظام التشغيل OS / 360، والذي يعطيني بعض التحفظات. إنه مليء بالنصائح على غرار "لقد فعلنا ذلك وفشلنا" و "كان علينا فعل هذا لأنه كان سيعمل." إن ملاحظات بروكس حول التقنيات التي فشلت لها أسس جيدة، لكن إدعاءاته بأن التقنيات الأخرى ستعمل هي تخمينية.. اقرأ الكتاب بشكل نقدي لفصل الملاحظات من التخمينات. لا يقلل هذا التحذير من القيمة الأساسية للكتاب. ولا يزال يتم الاستشهاد بهذا الكتاب في كتب الحوسبة، أكثر من أي كتاب آخر، وعلى الرغم من نشره في الأصل عام 1975، إلا أنه يبدو جديدًا اليوم. من الصعب قراءته دون قول "الحق عليّ!" كل صفحتين.

[1] cc2e.com/2813

معايير ذات صلة

IEEE Std 1058-1998: معيار لخطط إدارة مشروع البرمجيات.

IEEE Std 12207-1997: تكنولوجيا المعلومات – عمليات دورة حياة البرمجيات.

IEEE Std 1045-1992: معيار لمقاييس إنتاجية البرمجيات.

IEEE Std 1062-1998: الممارسة الموصى بها لتحصيل البرمجيات.

IEEE Std 1540-2001: معيار لعمليات دورة حياة البرمجيات – إدارة المخاطر.

IEEE Std 828-1998: معيار لخطط إدارة إعداد البرمجيات.

IEEE Std 1490-1998: دليل – اعتماد معيار PMI – دليل إلى هيئة المعرفة لإدارة المشاريع.

نقاط مفتاحية

- يمكن تحقيق ممارسات كتابة الشفرة الجيدة إما من خلال المعايير المطبقة أو من خلال اتباع نهج أكثر سلامًا.
- عند تطبيق إدارة الإعداد بشكل صحيح، فإنها تجعل مهام المبرمجين أسهل. ويشمل هذا على وجه الخصوص التحكم في التغيير.
- يعتبر التقدير الجيد للبرمجيات تحديًا كبيرًا. تستخدم مفاتيح النجاح أساليب متعددة، تضييق هامش تقديراتك أثناء عملك في المشروع، والاستفادة من البيانات لإنشاء التقديرات.
- القياس هو مفتاح النجاح في إدارة البناء. يمكنك العثور على طرق لقياس أي جانب من جوانب المشروع ما يكون أفضل من عدم قياسه على الإطلاق. القياس الدقيق هو المفتاح الأساسي لوضع جدول زمني دقيق، والتحكم بالجودة، وتحسين عملية التطوير الخاصة بك.
- المبرمجون والمدراء هم أشخاص، ويعملون بشكل أفضل عند معاملتهم على هذا النحو.

الفصل 29

التكامل

المحتويات[1]

مواضيع ذات صلة

- اختبار المطور: الفصل 22
- التصحيح: الفصل 23
- إدارة البناء: الفصل 28

يُشير المُصطلح "تكامل" إلى نشاط تطوير البرمجيات، الذي يمكنك فيه دمج مكونات برمجية منفصلة داخل نظام واحد. في المشاريع الصغيرة، قد يتألف التكامل من صباح واحد مقضي في تجميع عدد قليل من الصفوف معًا. أما في المشاريع الكبيرة، فمن الممكن أن يتألف من عدة أسابيع أو أشهر مقضية في ربط مجموعة من البرامج مع بعضها البعض. بغض النظر عن حجم المهمة، تُطبَّق قواعد مشتركة.

يتشابك موضوع التكامل مع موضوع تسلسل البناء. حيث يؤثر الترتيب الذي تنشئ فيه الصفوف أو المكونات على الترتيب الذي يمكنك دمجها فيه – فلا يمكنك دمج شيء لم يتم إنشاؤه بعد. إن كلا من التكامل وتسلسل عمليات البناء مواضيع مهمة. يتناول هذا الفصل كلا الموضوعين من وجهة نظر التكامل.

29. 1 أهمية منهج التكامل

في المجالات الهندسية، وبخلاف المجالات البرمجية، فإن أهمية التكامل السليم معروفة جيدًا. شهدت منطقة شمال غرب المحيط الهادئ، حيث أعيش، صورة دراماتيكية لمخاطر ضعف التكامل عندما انهار ملعب كرة القدم في جامعة واشنطن بشكل جزئي خلال البناء، كما هو موضح في الشكل 29-1.

[1] cc2e.com/2985



الشكل 29-1 انهارت إضافة ملعب كرة القدم في جامعة واشنطن لأنها لم تكن قوية بما يكفي لدعم نفسها أثناء عملية البناء. من المحتمل أن تكون قوية بما فيه الكفاية عند اكتمالها، ولكن تم بناؤها بترتيب خاطئ - خطأ التكامل.

لا يهم إن كان الملعب سيكون قويًا بما فيه الكفاية في الوقت الذي يتم فيه الانتهاء من بناءه؛ بل يجب أن يكون قويا بما فيه الكفاية في كل خطوة. إذا كنت بصدد إنشاء برمجية ودمجها بترتيب خاطئ، فذلك أصعب لكتابة الشفرة، وأصعب للاختبار وللتصحيح. وإذا لم يعمل أيا منها حتى يعمل الكل، فسيبدو كما لو أنه لن ينتهي بناءه أبدًا. كما أنه يمكن أن ينهار تحت تأثير ثقله أثناء البناء – فقد يبدو عدد الأخطاء تعجيزيًا، أو قد يكون التقدم غير ملحوظ، أو قد يكون التعقيد مفرطًا– حتى لو كان المنتج النهائي قد نجح.

يُنظر إلى التكامل أحيانًا على أنه نشاط اختبار، نظرًا لإتمامه بعد انتهاء مطور البرامج من اختبار المطور وبالتقاطع مع اختبار النظام. ومع ذلك، فإنه معقد بما فيه الكفاية، حيث ينبغي أن يُعتبر نشاطًا مستقلًا.



يمكنك أن تتوقع بعضا من هذه الفوائد من التكامل المُتأني:

- تشخيص أسهل للعيوب
- عيوب أقل
- سقالات أقل
- وقت أقل لأول مُنتج "شغال"
- جداول زمنية شاملة أقصر لعملية التطوير
- علاقات أفضل مع العملاء
- روح معنوية مُحسنة
- فرصة مُحسنة لاستكمال المشروع

- تقديرات أكثر موثوقية للجدول الزمني
- مزيد من الدقة في الإبلاغ عن حالة ما
- الجودة المُحسنة للشفرة
- عملية توثيق أقل

قد تبدو هذه مثل ادعاءات راقية لـ "ابن العم" المنسي لاختبار النظام، لكن حقيقة أنه تم التغاضي عنه على الرغم من أهميته هو بالضبط السبب في أن التكامل له فصله الخاص في هذا الكتاب.

2.29 تردد التكامل - على مراحل أم تزايدي؟

يتم دمج البرامج إما عن طريق النهج المرحلي أو التزايدي.

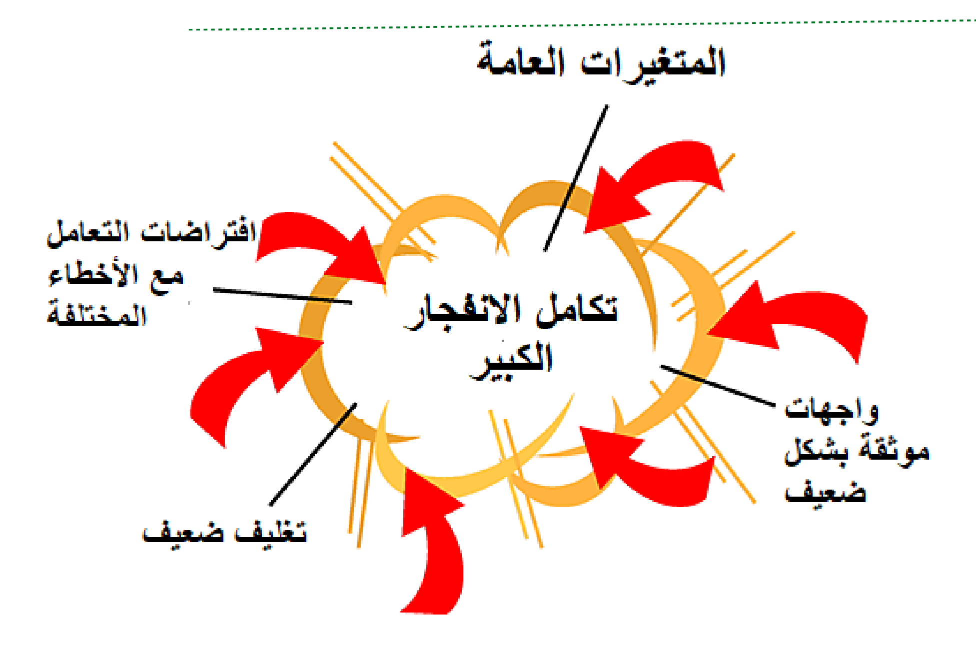
التكامل المرحلي

حتى سنوات قليلة مضت، كان التكامل المرحلي هو الأسلوب المتبع. حيث يتبع هذه الخطوات أو المراحل المحددة جيدًا:

- التصميم، وكتابة الشفرة، والاختبار، وتصحيح كل صف. هذه الخطوة تسمى "تطوير الوحدة".
- جمع الصفوف في نظام كبير ضخم ("دمج النظام").
- اختبار وتصحيح النظام بأكمله. وهذا ما يسمى "عدم تكامل النظام". (شكرا لميلير بيج جونز على هذه الملاحظة الذكية.)

تتمثل إحدى المشكلات المتعلقة بالتكامل المرحلي في أنه عندما يتم وضع الصفوف في نظام ما لأول مرة، فإن المشاكل الجديدة ستطفو على السطح حتما وأسباب المشكلات يمكن أن تكون في أي مكان. ونظرًا لأنه لديك عددًا كبيرًا من الصفوف التي لم تعمل معًا من قبل، فقد يكون المتهم بالمشاكل هو صف مُختبر بشكل سيئ، أو خطأ ناتج عن الواجهة بين صفين، أو خطأ ناتج عن التفاعل بين صفين. حيث جميع الصفوف مشكوك فيها.

ويزداد عدم اليقين بشأن موقع أي من المشاكل المحددة من حقيقة أن جميع المشاكل تظهر نفسها فجأة في وقت واحد. وهذا يدفعك إلى التعامل ليس فقط مع المشاكل الناجمة عن التفاعلات بين الصفوف، ولكن مع المشاكل التي يصعب تشخيصها لأن المشاكل نفسها تتفاعل مع بعضها البعض. لهذا السبب، هناك اسم آخر للتكامل المرحلي هو "تكامل الانفجار الكبير"، كما هو موضح في الشكل 29-2.



الشكل 29-2 يُطلق على الاندماج المرحلي تكامل "الانفجار الكبير" لسبب وجيه!

لا يمكن بدء التكامل المرحلي حتى وقت متأخر من المشروع، لبَعد اختبار المطور لجميع الصفوف. عندما يتم الجمع بين الصفوف في النهاية والأخطاء السطحية بالمحصلة، يذهب المبرمجين على الفور إلى وضع التصحيح المضطرب، بدلًا من الكشف المنهجي عن الأخطاء والتصحيح.

بالنسبة للبرامج الصغيرة – ليس للبرامج الصغيرة جدًا – قد يكون التكامل المرحلي أفضل نهج. إذا كان البرنامج يحتوي على صفين أو ثلاثة صفوف فقط، فقد يوفر التكامل المرحلي الوقت إذا كنت محظوظًا. لكن في معظم الحالات، هناك منهج آخر أفضل.

التكامل التزايدي

في التكامل التزايدي، تقوم بكتابة واختبار برنامج في أجزاء صغيرة، ومن ثم تدمج هذه الأجزاء في وقت واحد[1]. في هذا النهج للتكامل (جزء واحد في كل مرة)، اتّبع الخطوات التالية:

- تطوير جزء صغير وظيفي من النظام. يمكن أن يكون أصغر جزء وظيفي، أو الجزء الأصعب، أو جزء أساسي، أو مزيج ما. وثم اختباره وتصحيحه بشكل كامل. سيكون بمثابة هيكل عظمي لتعليق العضلات والأعصاب والجلد التي تشكل الأجزاء المتبقية من النظام.
- تصميم وكتابة شفرة واختبار وتصحيح صف.

[1] **إشارة مرجعية**: نوقشت الاستعارات المناسبة للتكامل التصاعدي في " برنامج Oyster Farming: نظام تراكم"، و "بناء البرمجيات"، كلاهما في القسم 3.2.

- دمج الصــف الجديد مع الهيكل العظمي. اختبار وتصــحيح تركيبة الهيكل العظمي مع الصــف الجديد. التأكد من أن التركيبة تعمل قبل إضافة أية صفوف جديدة أخرى. إذا كان لا يزال يتعين القيام بالعمل، كرر العملية التي تبدأ في الخطوة 2.

أحيانا، قد ترغب في دمج وحدات أكبر من صف واحد. على سبيل المثال، إذا تم اختبار أحد المكونات تماما، ووضع كل صف من صفوفه من خلال تكامل صغير، فيمكنك دمج المكون بالكامل أثناء قيامك بتكامل تزايدي. كلما أضفت أجزاءً إليه، ينمو النظام ويكتسب الزخم بنفس الطريقة التي تنمو بها كرة الثلج وتكتسب دفعًا عندما تتحرك إلى أسفل التل، كما هو موضح في الشكل 29-3.



الشكل 29-3 يساعد التكامل التزايدي المشروع على بناء الزخم، مثل كرة الثلج المتدحرجة إلى أسفل التل.

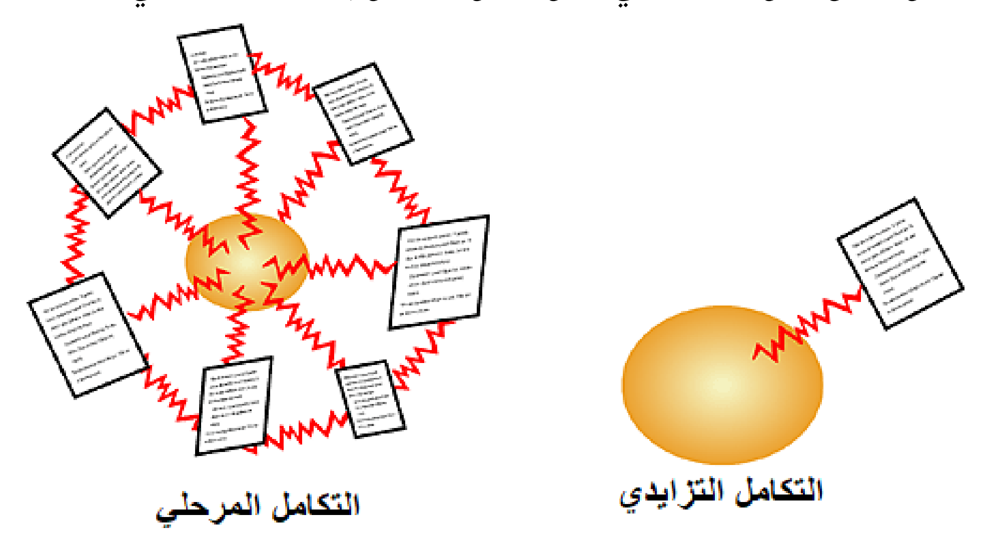
فوائد التكامل التزايدي

يقدم النهج التزايدي العديد من المزايا مقارنة بالنهج المرحلي التقليدي بغض النظر عن أي استراتيجية تصاعدية تستخدمها:

بيانات مثبتة

من السهل إيجاد الأخطاء. عندما تظهر مشاكل جديدة خلال التكامل التزايدي، فمن الواضح أن الصف الجديد متورط في هذه المشاكل. إما أن واجِهته مع بقية البرنامج تحتوي على خطأ أو أن تفاعله مع صف مدمج مسبقًا ينتج خطأ. في كلتا الحالتين، كما هو مقترح في الشكل 29-4، فأنت تعرف بالضبط أين تبحث. علاوة على ذلك، لمجرد أن لديك مشاكل أقل في وقت واحد، فإنك تقلل من خطر أن تتفاعل المشاكل المتعددة، أو أن تُخفي مشكلة ما مشكلة أخرى. كلما زادت أخطاء الواجهة التي تميل إلى امتلاكها، كلما زادت فائدة هذا التكامل التزايدي في مشاريعك. كشف حساب عدد الأخطاء لأحد المشاريع أن 39 في المئة كانت أخطاء في الواجهة البينية فيما بين الوحدات (باسيلي وبريكون، 1984). نظرًا لأن المطورين ينفقون في العديد

من المشاريع ما يصل إلى 50 بالمئة من وقتهم في عملية تصحيح الأخطاء، فإن تحقيق أقصى قدر من فعالية تصحيح الأخطاء عن طريق جعل الأخطاء في مكان يسهل الوصول إليه يوفر فوائد في الجودة والإنتاجية.



الشكل 29-4 في التكامل المرحلي، تقوم بدمج العديد من المكونات في وقت واحد بحيث يصعب معرفة مكان الخطأ. قد يكون في أي من المكونات أو في أي من اتصالاتهم. أما في التكامل التزايدي، يكون الخطأ عادةً إما في المكون الجديد أو في الاتصال بين المكون الجديد والنظام.

ينجح النظام في وقت مبكر من المشروع. عندما يتم دمج الشفرة وتشغيلها، حتى إذا كان النظام غير صالح للاستخدام، فمن الواضح أنه سيكون صالح للاستخدام قريبًا. مع التكامل التزايدي، يرى المبرمجون النتائج المبكرة من عملهم، لذلك فإن معنوياتهم أفضل مما ستكون عليه عندما يشكون في أن مشروعهم لن يلفظ أنفاسه أبدًا.

يمكنك الحصول على مراقبة مُحسنة للتقدم. عند التكامل بشكل متكرر تكون الميزات الموجودة وغير الموجودة واضحة. سيكون لدى الإدارة إحساس أفضل بالتقدم من رؤية 50 بالمئة من قدرة النظام على العمل أكثر من سماع أن التشفير "اكتمل 99 بالمئة".

سوف تحسن علاقات العملاء. إذا كان التكامل المتكرر له تأثير على معنويات المطور، فإنه يؤثر أيضًا على معنويات العملاء. يُحب العملاء علامات التقدم، والبناءات الإضافية التي تؤمن علامات التقدم بشكل متكرر.

يتم اختبار وحدات النظام بشكل أكثر كمالا. يبدأ التكامل في وقت مبكر من المشروع. حيث يدمج كل صف في الوقت الذي يتم فيه تطويره، بدلاً من انتظار حفلة صاخبة من الدمج في النهاية. في كلتا الحالتين يتم اختبار الصفوف باختبار المطور، ولكن ينفّذ كل صف كجزء من النظام الكلي في كثير من الأحيان باستخدام التكامل التزايدي أكثر من التكامل المرحلي.

يمكنك بناء النظام بجدول زمني للتطوير أقصر. إذا تم التخطيط للتكامل بعناية، يمكنك تصميم جزء من النظام بينما يتم كتابة شفرة لجزء آخر. لا يقلل ذلك من العدد الإجمالي لساعات العمل المطلوبة لتطوير التصميم الكامل والشفرة، ولكنه يسمح ببعض العمل المتوازي في وقت واحد، وهذه ميزة عندما يكون وقت تنفيذ المشروع قليل.

يدعم ويشجع التكامل التزايدي الاستراتيجيات التزايدية الأخرى. مزايا التزايد المُطبقة على التكامل هي قمة جبل جليدي.

3 .28 استراتيجيات التكامل التزايدي

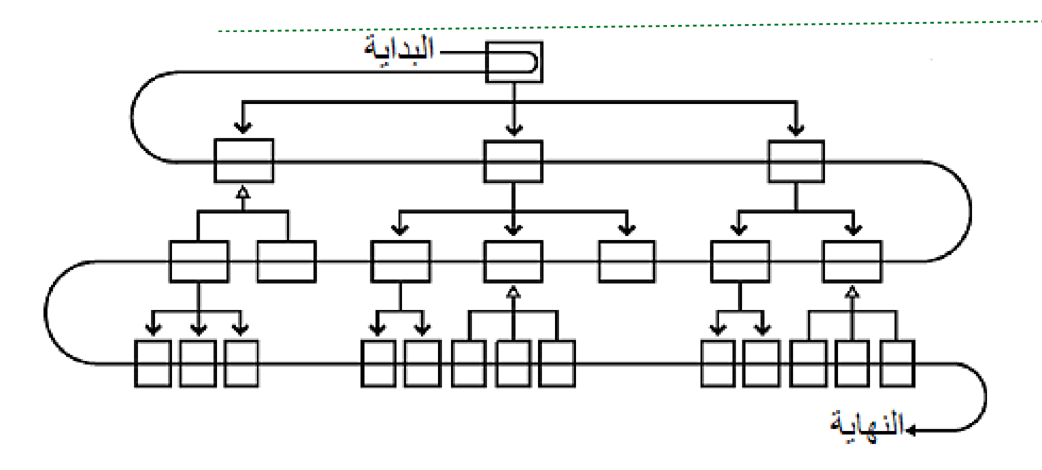
مع التكامل المرحلي، لن تحتاج إلى تخطيط ترتيب مكونات المشروع التي يتم بناؤها. حيث تدمج جميع المكونات في الوقت نفسه، بحيث يمكنك بناؤها بأي ترتيب طالما ستكون جاهزة قبل اليوم المهم (يوم انتهاء المشروع).

ولكن مع التكامل التزايدي، عليك أن تخطط بعناية أكثر. ستدعو معظم الأنظمة إلى دمج بعض المكونات قبل دمج المكونات الأخرى. يؤثر التخطيط للتكامل بهذه الطريقة على التخطيط للبناء. حيث يجب أن يدعم الترتيب الذي يتم به إنشاء المكونات الترتيب الذي سيتم دمجها به.

تأتي استراتيجيات ترتيب التكامل في مجموعة متنوعة من الأشكال والأحجام، ولا تعتبر واحدة منهم الأفضل في جميع الحالات. يختلف أفضل نهج للتكامل من مشروع إلى آخر، وأفضل حل هو دائمًا هو الحل الذي تقوم بإنشائه لتلبية المتطلبات المحددة لمشروع معين. إن معرفة النقاط على خط الأعداد المنهجي سوف تعطيك رؤية للحلول الممكنة.

التكامل من الأعلى إلى الأسفل

في التكامل من الأعلى إلى الأسفل، تتم كتابة الصف الموجود أعلى التسلسل الهرمي أولًا. الجزء العلوي هو النافذة الرئيسية، وحلقة التحكم في التطبيقات، والكائن الذي يحتوي على () main في Java، و() WinMain لبرمجة مايكروسوفت ويندوز، أو ما شابه. أما الجزء السفلي فلابد من كتابته لتنفيذ الصف العلوي. بعد ذلك، عندما يتم دمج الصفوف من الأعلى إلى الأسفل، يتم استبدال الصفوف السفلية بصفوف حقيقية. يستمر هذا النوع من التكامل كما هو موضح في الشكل 29-5.



الشكل 29-5 في التكامل من الأعلى إلى الأسفل، تقوم بإضافة صفوف في الأعلى أولًا، وأخيرًا في الأسفل.

إن أحد الجوانب المهمة في التكامل من الأعلى إلى الأسفل هو أنه يجب تحديد الواجهات بين الصفوف بعناية. فالأخطاء الأكثر صعوبة في التصحيح ليست هي تلك التي تؤثر على الصفوف الفردية ولكن تلك التي تنشأ من التفاعلات الدقيقة بين الصفوف. يمكن أن يساعد تحديد مواصفات الواجهة بدقة في الحد من المشكلة. تحديد الواجهة ليس بنشاط تكامل، ولكن التأكد من أن الواجهات قد تم تحديدها جيدًا هو أحد نشاطات التكامل.

بالإضافة إلى المزايا التي تحصل عليها من أي نوع من التكامل التزايدي، فإن ميزة التكامل من الأعلى إلى الأسفل هي أن منطق التحكم في النظام يتم اختباره مبكرًا. تتحمل جميع الصفوف في الجزء العلوي من التسلسل الهرمي الكثير من العبء، لذلك فإن مشاكل التصميم المفاهيمية الضخمة تظهر بسرعة.

ميزة أخرى للتكامل من الأعلى إلى الأسفل وهي أنه إذا كنت تخطط بعناية، فيمكنك إكمال نظام العمل بشكل جزئي في وقت مُبكر من المشروع. إذا كانت أجزاء واجهة المستخدم في الأعلى، فيمكنك الحصول على واجهة أساسية تعمل بسرعة وتوضيح التفاصيل لاحقًا. حيث يفيد الحصول على شيء مرئي يعمل في وقت مبكر الروح المعنوية لكل من المستخدمين والمبرمجين على حد سواء.

يُتيح لك التكامل التصاعدي من الأعلى إلى الأسفل أيضًا بدء كتابة الشفرة قبل اكتمال تفاصيل تصميم المستوى المنخفض. بمجرد أن يتم دفع التصميم إلى مستوى منخفض إلى حد ما من التفاصيل في جميع المناطق، يمكنك البدء في تنفيذ ودمج الصفوف في المستويات الأعلى دون انتظار كل التفاصيل.

على الرغم من هذه المزايا، فإن التكامل من الأعلى إلى الأسفل عادةً ما يتضمن عيوبًا أكثر إزعاجًا مما قد ترغب في مواجهته. يترك التكامل الصافي من الأعلى إلى الأسفل استخدام واجهات النظام صعبة حتى النهاية. إذا كانت واجهات النظام مشوبة بالأخطاء أو تشكل مشكلة أداء، فعادة ما ترغب في الوصول إليها قبل وقت طويل من نهاية المشروع. إنه من المعتاد أن تتدفق مشكلة المستوى المنخفض إلى أعلى النظام، مما يتسبب في

تغييرات عالية المستوى ويقلل من فائدة أعمال التكامل السابقة. قلّل من مشكلة تدفق المشاكل من خلال اختبار مطور مبكر حذر، وتحليل أداء للصفوف التي تُنفذ واجهات النظام.

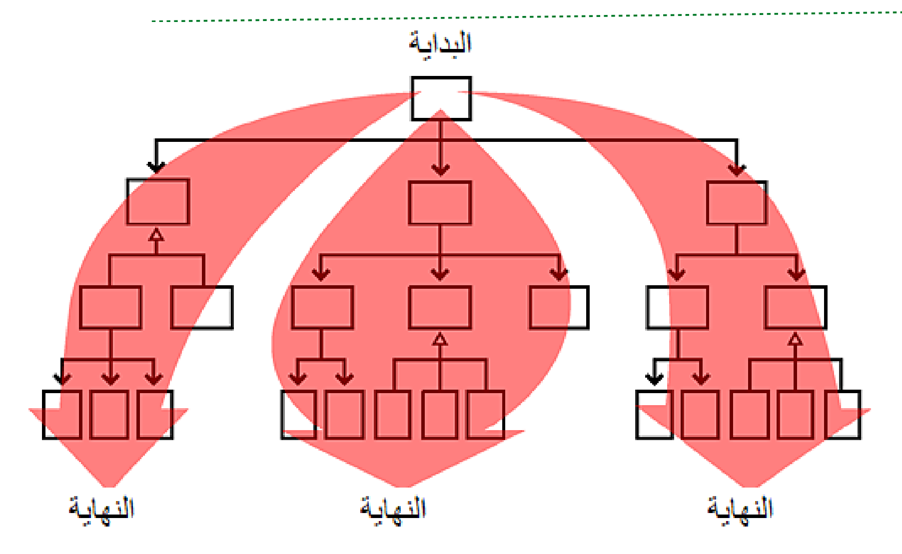
هناك مشكلة أخرى تتعلق بالتكامل من الأعلى إلى الأسفل، وهي أنك تحتاج إلى كمية كبيرة من الأجزاء السفلية للدمج من الأعلى إلى الأسفل. فالعديد من الصفوف منخفضة المستوى لم تدمج بعد، وهو ما يعني ضمنًا أن هناك حاجة إلى عدد كبير من مكونات الطبقة السفلى خلال الخطوات الوسيطية في الدمج. تُعتبر الطبقة السفلى إشكالية في ذلك، كشفرة الاختبار، فمن المرجح أن تحتوي على أخطاء أكثر من شفرة الإنتاج المصممة بعناية. تؤدي الأخطاء الموجودة في المكونات الجديدة للطبقة السفلى التي تدعم صف جديد إلى إحباط الغرض من التكامل التزايدي، وهو الحد من مصدر الأخطاء لصف جديد واحد.

يكاد يكون التكامل من الأعلى إلى الأسفل هو أيضا من المستحيل تنفيذه بشكل صافي[1]. في التكامل من الأعلى إلى الأسفل الذي ينجزه الكتاب، تبدأ من الأعلى - يطلق عليه المستوى 1 - ثم تُدمج جميع الصفوف في المستوى التالي (المستوى 2). عند دمج جميع الصفوف من المستوى 2، وليس قبل ذلك، يمكنك دمج الصفوف من المستوى 3. الصلابة في التكامل الصافي من الأعلى إلى الأسفل هي أمر اعتباطي تمامًا. من الصعب تخيل أي شخص يختار عناء استخدام التكامل الصافي من الأعلى إلى الأسفل. يستخدم معظم الأشخاص أسلوبًا هجينًا، مثل التكامل من الأعلى إلى الأسفل في الأقسام بدلاً من ذلك.

وأخيرًا، لا يمكنك استخدام التكامل من الأعلى إلى الأسفل إذا لم تكن مجموعة الصفوف في الأعلى موجودة. في العديد من الأنظمة التفاعلية، يكون تحديد موقع "الأعلى" أمر شخصي. ففي العديد من الأنظمة، تكون واجهة المستخدم هي الأعلى. أما في الأنظمة الأخرى، يكون `main()` هو الأعلى.

هناك بديل جيد للتكامل الصافي من الأعلى إلى الأسفل هو نهج الشريحة الرأسية الموضح في الشكل 29-6. في هذا النهج، يتم تنفيذ النظام من الأعلى إلى الأسفل في الأقسام، وربما يتم التعامل مع مجالات الوظائف واحدًا تلو الآخر ثم تنتقل إلى المنطقة التالية.

[1] **إشارة مرجعية:** يرتبط التكامل من الأعلى إلى الأسفل بالتصميم من الأعلى إلى الأسفل بالاسم فقط. للحصول على تفاصيل حول التصميم من الأعلى إلى الأسفل، راجع "مقاربات التصميم من الأعلى إلى الأسفل ومن الأسفل إلى الأعلى" في القسم 4.5.

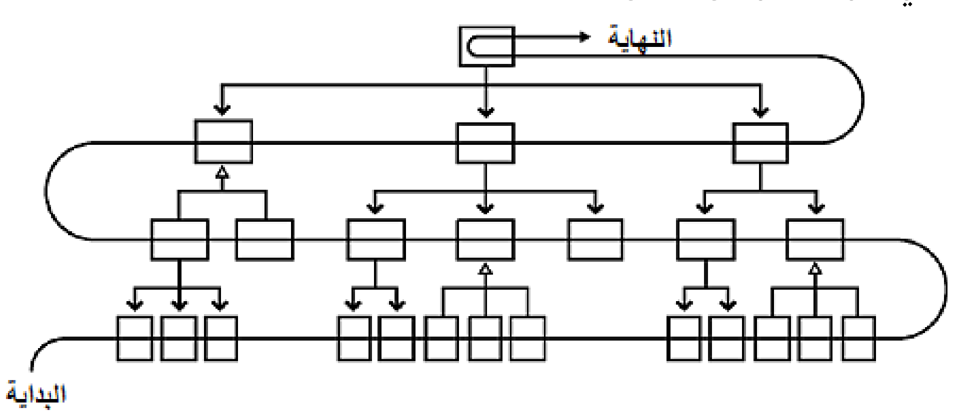


الشكل 29-6 كبديل للتقدم الدقيق من الأعلى إلى الأسفل، يمكنك الدمج من الأعلى إلى الأسفل في الشرائح الرأسية.

على الرغم من أن التكامل الصافي من الأعلى إلى الأسفل غير قابل للتطبيق، إلا أن التفكير فيه سيساعدك على اتخاذ قرار بشأن النهج العام. تنطبق بعض الفوائد والمخاطر التي تنطبق على النهج الصافي من الأعلى إلى الأسفل، وبشكل أقل وضوحًا، على النُهج المرنة من الأعلى إلى الأسفل مثل تكامل الشريحة الرأسية، لذا ضعها في اعتبارك.

التكامل من الأسفل إلى الأعلى

في التكامل من الأسفل إلى الأعلى، تقوم بكتابة ودمج الصفوف في أسفل التسلسل الهرمي أولًا. إن إضافة الصفوف ذات المستوى المنخفض واحد في كل مرة وليس كلها في وقت واحد هو ما يجعل التكامل من الأسفل إلى الأعلى استراتيجية تكامل تزايدي. تكتب برامج اختبار لاختبار الصفوف منخفضة المستوى في البداية ومن ثم تقوم بإضافة صفوف إلى السقالات التجريبية عند تطويرها. عند إضافة صفوف المستوى الأعلى، يمكنك استبدال صفوف برنامج التشغيل بأخرى حقيقية. يُوضح الشكل 29-7 الترتيب الذي يتم فيه دمج الصفوف في المنهج التزايدي (من الأسفل إلى الأعلى).



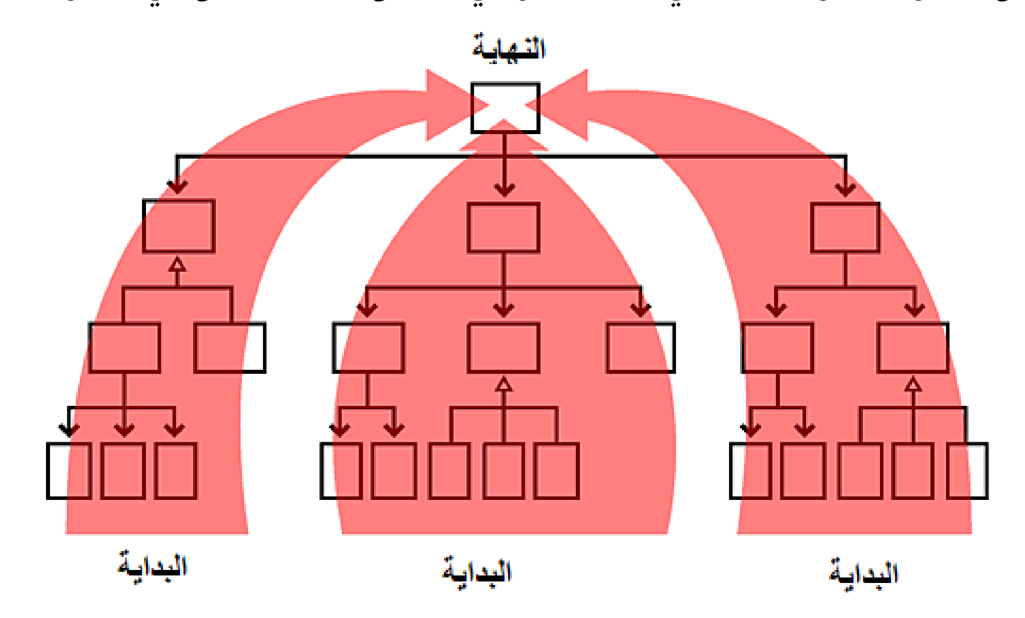
الشكل 29-7 في التكامل من الأسفل إلى الأعلى، تقوم بدمج الصفوف في أسفل أولًا، وفي الجزء العلوي أخيرًا.

يوفر التكامل من الأسفل إلى الأعلى مجموعة محدودة من مزايا التكامل التزايدي. إنه يحد من المصادر المحتملة للخطأ بسبب دمج صف واحد، لذلك من السهل تحديد الأخطاء. ويمكن أن يبدأ التكامل في وقت مبكر من المشروع. أيضا يُظهر تكامل "من الأسفل الى الأعلى" مشاكل واجهات النظام بشكل مبكر، نظرًا لأن قيود النظام تحدد غالبًا ما إذا كنت تستطيع تحقيق أهداف النظام، للتأكد من أن تنفيذ النظام لمجموعة كاملة من تمارين الجمباز أمر يستحق العناء.

المشكلة الرئيسية في التكامل من الأسفل إلى الأعلى هي أنه يترك التكامل بين واجهات النظام الرئيسية، عالية المستوى حتى الآخر. إذا كان لدى النظام مشاكل في التصميم المفاهيمي في المستويات الأعلى، فلن يعثر عليها البناء حتى يتم تنفيذ جميع الأعمال التفصيلية. إذا كان يجب تغيير التصميم بشكل ملحوظ، فقد تهمل بعض الأعمال منخفضة المستوى.

تتطلب طريقة التكامل من" الأسفل الى الأعلى " منك اكمال تصميم النظام بشكل تام قبل البدء بعملية التكامل، فاذا لم تقم بذلك، فإن الافتراضات التي لا يتم التحكم بها في التصميم ربما تكون متواجدة بعمق في الشفرة ذات المستوى المنخفض، مما يؤدي إلى وضع غير ملائم لتصميم طبقات عالية المستوى للتغلب على المشاكل في المستوى المنخفض. ترك التفاصيل منخفضة المستوى تقود تصميم صفوف المستوى الأعلى يناقض مبادئ إخفاء المعلومات والتصميم كائني التوجه. إن مشاكل تكامل صفوف المستوى الأعلى ليست سوى قطرة في عاصفة مطيرة مقارنة بالمشاكل التي ستواجهها إذا لم تكمل تصميم الصفوف عالية المستوى قبل البدء في كتابة تفاصيل شفرة المستوى المنخفض.

كما هو الحال مع التكامل من الأعلى إلى الأسفل، يعد التكامل الصافي من الأسفل إلى الأعلى أمرًا نادرًا، ويمكنك استخدام منهج هجين بدلاً من ذلك، بما في ذلك التكامل في الشرائح كما هو موضح في الشكل 29-8.

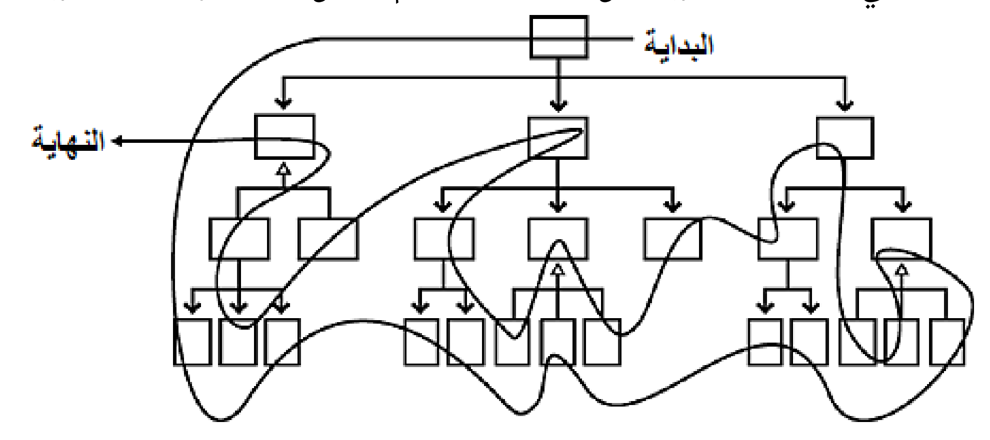


الشكل 29-8 كبديل للتقدم الصافي من الأسفل إلى الأعلى، يمكنك التكامل من الأسفل إلى الأعلى في الأقسام. هذا يطمس الخط الفاصل بين التكامل من الأسفل إلى الأعلى والتكامل الموجه بالميزات، والذي يرد وصفه لاحقًا في هذا الفصل.

تكامل الساندويتش

دفعت المشاكل المتعلقة بالتكامل الصافي من الأعلى إلى الأسفل والتكامل الصافي من الأسفل إلى الأعلى بعض الخبراء إلى التوصية بنهج الساندويتش (مايرز 1976). يمكنك أولاً دمج صفوف كائنات الأعمال عالية المستوى في أعلى التسلسل الهرمي. ومن ثم تقوم بدمج صفوف واجهة الجهاز وصفوف المرافق المستخدمة على نطاق واسع في الجزء السفلي. هذه الصفوف عالية المستوى ومنخفضة المستوى هي خبز الساندويتش.

تقوم بترك صفوف الطبقة المتوسطة حتى وقت لاحق. وهذه تُشكل اللحوم والجبن والطماطم من الساندويتش. وإذا كنت نباتياً، فقد تُشكل التوفو وبراعم الفاصولياء الخاصة بالساندويتش، لكن مؤلف تكامل الساندويتش صامت في هذه النقطة - ربما كان فمه ممتلئًا. يقدم الشكل 29-9 توضيحًا لمقاربة الساندويتش.



الشكل 29-9 في تكامل الساندويتش، تقوم بدمج الصفوف ذات المستوى الأعلى والمستعملة على نطاق واسع من المستوى الأسفل أولاً، وتقوم بحفظ الصفوف المتوسطة للمستوى الأخير.

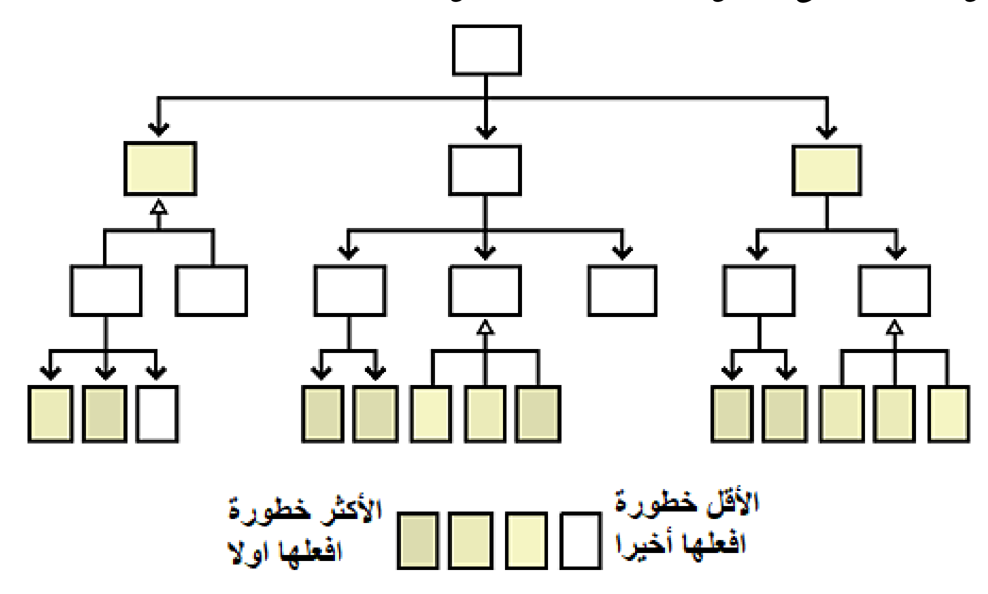
يتجنب هذا النهج صلابة التكامل الصافي من الأسفل إلى الأعلى أو من الأعلى إلى الأسفل. فهو يدمج الصفوف الأكثر إثارة للمشاكل في المقام الأول ولديه القدرة على تقليل كمية السقالات التي ستحتاج إليها. إنه نهج واقعي وعملي. النهج التالي مماثل ولكن يؤكد على أمور مختلفة.

التكامل الموجه بالمخاطر

ويسمى التكامل الموجه بالمخاطر أيضًا بـ "تكامل الجزء الصعب أولا". إنه يشبه تكامل الساندويتش في أنه يسعى إلى تجنب المشاكل المتأصلة في التكامل الصافي من الأعلى إلى الأسفل أو في التكامل الصافي من الأسفل إلى الأعلى. من قبيل الصدفة، فإنه يميل أيضا إلى دمج الصفوف في الأعلى وفي الأسفل أولًا، محافظًا على صفوف المستوى المتوسط للآخر. لكن الدافع هنا، على كل حال، مُختلف.

في التكامل الموجه بالمخاطر، يمكنك تحديد مستوى المخاطر المرتبطة بكل صف. عليك أن تقرر أية الأجزاء الأكثر تحديًا لتنفيذها، وتقوم بتنفيذها أولًا. تشير التجربة إلى أن واجهات المستوى الأعلى محفوفة بالمخاطر، لذا فهي غالبًا ما تكون في أعلى قائمة المخاطر. تعتبر واجهات النظام، الموجودة عادةً في المستوى السفلي للتسلسل الهرمي، خطرة أيضًا، لذا فهي أيضًا في أعلى قائمة المخاطر. بالإضافة إلى ذلك، قد تكون على دراية بأن الصفوف الموجودة في المنتصف ستكون صعبة. ربما يطبق الصف خوارزمية غير مفهومة بشكل جيد أو لديه أهداف أداء طموحة. ويمكن أيضا تحديد هذه صفوف على أنها مخاطر عالية وتُدمج في وقت مبكر نسبيًا.

ما تبقى من الشفرة، والأشياء السهلة، يمكن أن تنتظر حتى وقت لاحق. ربما يتضح أن بعضها أصعب مما ظننته، لكن هذا لا يمكن تجنبه. يُوضح الشكل 29-10 مثالًا للتكامل الموجه بالمخاطر.



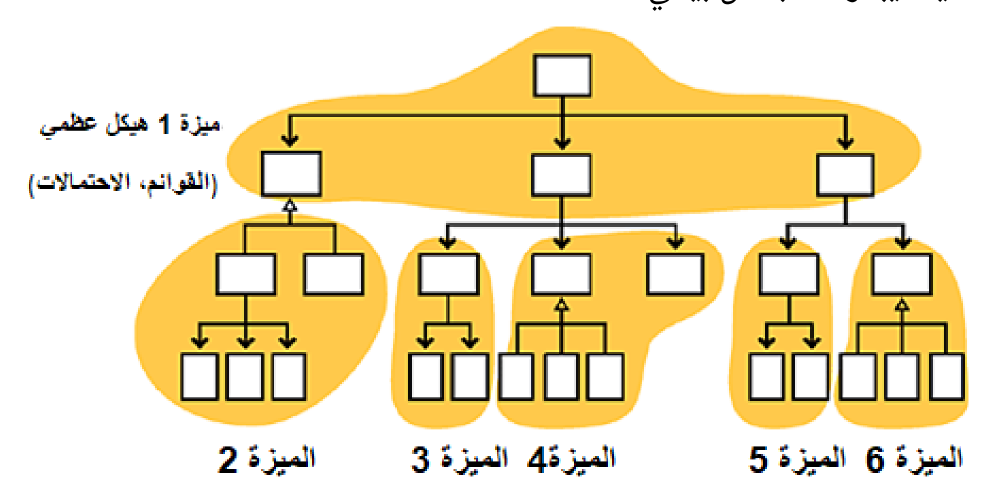
الشكل 29-10 في التكامل الموجه بالمخاطر، يمكنك دمج الصفوف التي تتوقع أن تكون الأكثر إثارة للقلق أولاً؛ ومن ثم تقوم بتنفيذ صفوف أسهل في وقت لاحق.

التكامل الموجه بالميزات

نهج آخر هو دمج ميزة واحدة في كل مرة. لا يشير المصطلح "ميزة" إلى أي شيء وهمي، مجرد تابع محدد للنظام الذي تُكامله. إذا كنت تكتب معالج كلمات، فالميزة قد تكون عرض إحدى المكوّنات أسفل الشاشة أو إعادة تنسيق المستند تلقائيًا - شيء من هذا القبيل.

عندما تكون الميزة المراد دمجها أكبر من صف واحدة، تكون "الزيادة" في التكامل التزايدي أكبر من صف واحد. يُقلّل هذا من فائدة التدرج قليلاً لكونه يقلل من يقينك حول مصدر الأخطاء الجديدة، ولكن إذا كنت قد اختبرت تمامًا الصفوف التي تنفذ الميزة الجديدة قبل دمجها، فهذا مجرد عيب صغير. يمكنك استخدام استراتيجيات التكامل التزايدي بشكل متكرر من خلال دمج القطع الصغيرة لتشكيل ميزات ثم دمج الميزات بشكل متزايد لتكوين نظام.

ستحتاج عادة إلى البدء بهيكل عظمي اخترته لقدرته على دعم الميزات الأخرى. في نظام تفاعلي، قد تكون الميزة الأولى هي نظام القائمة التفاعلية. يُمكنك تعليق بقية الميزات على الميزة التي تقوم بدمجها أولًا. يوضح الشكل 29-11 كيف يبدو هذا بشكل بياني.



الشكل 29-11 في الدمج الموجه بالميزات، تقوم بدمج الصفوف في المجموعات التي تشكل ميزات قابلة للتحديد — عادةً، ولكن ليس دائمًا، صفوف متعددة في كل مرة.

تتم إضافة المكونات في "أشجار الميزات"، وهي مجموعات هرمية من الصفوف التي تشكل ميزة. يكون التكامل أسهل إذا كانت كل ميزة مستقلة نسبيًا، وربما تستدعي نفس شفرة المكتبة منخفضة المستوى مثل الصفوف الميزات الأخرى، ولكن لا يوجد استدعاءات إلى شفرة متوسطة المستوى مشتركة مع ميزات أخرى. (لا تظهر صفوف المكتبة المشتركة منخفضة المستوى في الشكل 29-11).

يوفر التكامل الموجه بالميزات ثلاث مزايا رئيسية. أولًا، أنه يزيل السقالات لكل شيء تقريبًا باستثناء صفوف المكتبة منخفضة المستوى. قد يحتاج الهيكل العظمي إلى سقالة صغيرة، أو قد لا تعمل بعض أجزاء الهيكل العظمي ببساطة حتى يتم إضافة ميزات معينة. على كل حال، عندما تكون كل ميزة معلقة على الهيكل، لا توجد حاجة لسقالات إضافية. ونظرًا لأن كل ميزة مستقلة، فإن كل ميزة تحتوي على كل الشفرة الداعمة التي تحتاج إليها.

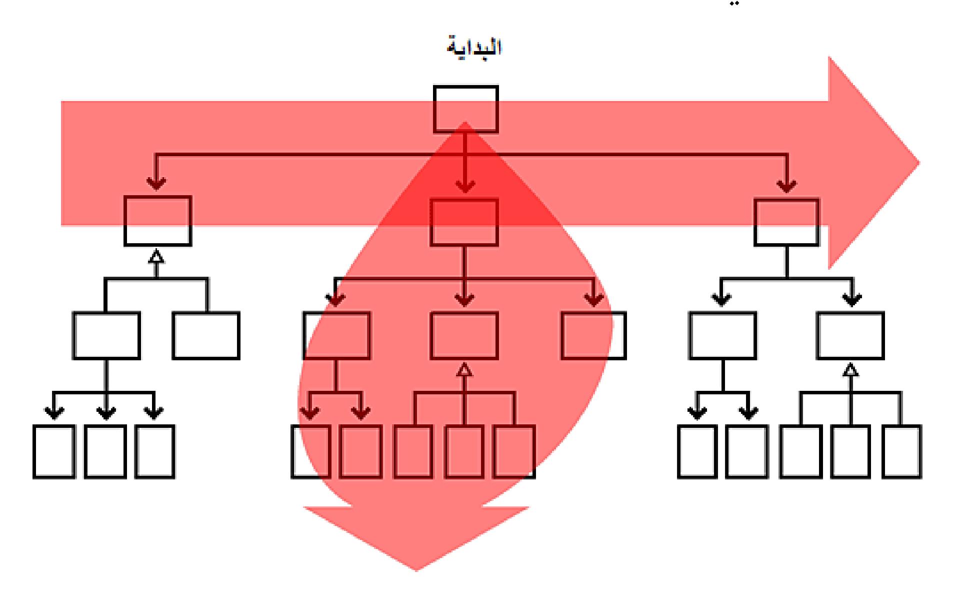
الميزة الرئيسية الثانية هي أن كل ميزة متكاملة جديدة تجلب إضافة إضافية في الوظائف. حيث يقدم هذا دليلًا على أن المشروع يتحرك بثبات إلى الأمام. كما أنه يُنشئ برنامجًا وظيفيًا يمكنك توفيره لعملائك للتقييم، أو يمكنك إصداره مسبقًا وبقدر أقل من الوظائف عما كان مخططًا له في الأصل.

الميزة الثالثة هي أن التكامل الموجه نحو الميزات يعمل بشكل جيد مع التصميم غرضي التوجه. تميل الكائنات إلى مقابلة جيدة للميزات، مما يجعل التكامل الموجه نحو الميزات خيارًا طبيعيًا للأنظمة غرضية التوجه.

من الصعب السعي لتحقيق تكامل موجه نحو الميزات كتكامل من الأعلى إلى الأسفل أو من الأسفل إلى الأعلى. عادةً، يجب دمج بعض الشفرة المنخفضة المستوى قبل التمكن من دمج بعض الميزات الهامة.

التكامل على شكل T

ويُسمى النهج النهائي الذي غالبًا ما يعالج المشاكل المرتبطة بالتكامل من الأعلى إلى الأسفل والتكامل من الأسفل إلى الأعلى "التكامل على شكل حرف T ". وفي هذا النهج، يتم اختيار شريحة واحدة محددة من أجل التطوير والتكامل المُبكر. وينبغي لهذه الشريحة أن تُنفذ النظام من طرف إلى طرف وأن تكون قادرة على استبعاد أية مشاكل رئيسية في افتراضات تصميم النظام. بمجرد أن يتم تنفيذ هذه الشريحة الرأسية – وتصحيح أية مشاكل مرتبطة بها - يمكن تطوير العرض الشامل للنظام (مثل نظام القائمة في تطبيق سطح المكتب). وكثيرًا ما يتم الجمع بين هذا النهج، الموضح في الشكل 29-12، مع نهج التكامل الموجه بالمخاطر أو الموجه بالميزات.



الشكل 29-12 في التكامل على شكل T، تقوم بإنشاء شريحة عميقة من النظام ودمجها للتحقق من الافتراضات المعمارية، ومن ثم تقوم ببناء ودمج عرض النظام لتوفير إطار عمل لتطوير الوظائف المتبقية.

ملخص نُهج التكامل

من الأسفل إلى الأعلى، ومن الأعلى إلى الأسفل، وتكامل السندويش، والموجهة بالمخاطر، والموجهة بالميزات، وعلى شكل حرف T- هل تشعر بأن الناس يقومون بإعداد هذه الأسماء أثناء سيرهم؟ نعم إنهم يقومون بهذا. لا تُعتبر أي من هذه الطرق إجراءات قوية يجب عليك اتباعها بطريقة منهجية من الخطوة 1 إلى الخطوة 47، ثم تعلن أنك أتممت ذلك. إنها مثل نُهج تصميم البرمجيات، فهي أكثر استدلالية من الخوارزميات. وبدلًا من اتباع أي إجراء دوغماتي، يمكنك الخروج من خلال وضع استراتيجية فريدة مُصممة خصيصًا لمشروعك المحدد.

4.29 البناء اليومي واختبار الدخان

تنويه: البناء المقصود هنا غير البناء الذي يناقشه هذا الكتاب. البناء المقصود في هذا القسم هو العملية build للمشروع قيد التطوير والمعروفة جيدا للمبرمجين. والتي تؤدي إلى إنشاء ملفات تنفيذية أو ملفات مكتبات الربط الديناميكي dll. يمكن تشغيلها خارج بيئة التطوير.

مهما كانت استراتيجية التكامل التي تختارها، فإن المقاربة الجيدة لتكامل البرنامج هي "البناء اليومي واختبار الدخان"[1]. كل ملف يُترجم ويربط ويدمج في برنامج قابل للتنفيذ كل يوم، ثم يوضع البرنامج خلال "اختبار الدخان"، وهو فحص بسيط نسبيًا لمعرفة ما إذا كان المنتج "يُدخن" عند تشغيله.

تُنتج هذه العملية البسيطة العديد من الفوائد الهامة. فهي تُقلل من مخاطر الجودة المنخفضة، وهي مخاطر تتعلق بخطر التكامل غير الناجح أو الإشكالي. فعن طريق اختبار دخان يومي لجميع الشفرات، يتم منع مشاكل الجودة من السيطرة على المشروع. فأنت تجلب النظام إلى حالة جيدة معروفة، ثم تحتفظ بها هناك. أنت بذلك ببساطة لا تسمح له بالتدهور إلى الحد الذي يمكن أن تحدث فيه مشكلات جودة مستهلكة للوقت.

تدعم هذه العملية أيضا التشخيص الأسهل للعيوب. عندما يتم بناء المنتج واختباره كل يوم، فمن السهل تحديد سبب فشل عمل المُنتج في أي يوم مُعطى. إذا كان المنتج يعمل في اليوم 17 وفشل في العمل في اليوم 18، فإن شيئًا حدث بين اليومين قد عطل المنتج.

يُحسن هذا من الروح المعنوية. حيث توفر رؤية المُنتَج يعمل دفعة لا تصدق إلى الروح المعنوية. يكاد لا يهم ما يفعله المُنتَج. يمكن أن يكون المطورون متحمسين فقط لرؤيته يُظهر مستطيلاً! أو أكثر قليلًا، وأن يعمل المنتج كل يوم، وهذا يحافظ على معنويات عالية.

أحد الآثار الجانبية للتكامل المتكرر هو أنه يبرز العمل الذي يمكن أن يتراكم بطريقة أخرى حتى يظهر بشكل غير متوقع في نهاية المشروع. يمكن أن يتحول تراكم العمل غير المُعبد إلى حفرة قاتلة في نهاية المشروع تستغرق أسابيع أو أشهر للخروج منها.

---

[1] **قراءة متعمقة**: تم اقتباس الكثير من هذه المناقشة من الفصل 18 من التطور السريع (ماكونيل 1996). إذا قرأت هذه المناقشة، فيمكنك الانتقال إلى قسم "التكامل المستمر".

هامش: اختبار الدخان (Smoke Test): وهي مجموعة من حالات الاختبارات التي يقوم بإعدادها فريق الفحص، بحيث تختبر المهام الأساسية للتطبيق وفي حال عدم عمل أي منها يتم الإرجاع إلى فريق التطوير لحل المشاكل، وهذا النوع غير مكلف وتقوم به شركة مايكروسوفت بشكل يومي عند أي تجديد على المنتجات

في بعض الأحيان تشعر الفِرق التي لم تستخدم عملية البناء اليومية أن البناء اليومي يؤدي إلى إبطاء تقدمها إلى حد زحف الحلزون. ما يحدث حقًا هو أن البناء اليومي يعمل بشكل أكثر ثباتًا طوال المشروع، ويحصل فريق المشروع فقط على صورة أكثر دقة عن مدى السرعة التي كان يعمل بها طوال الوقت.

فيما يلي بعض التفاصيل الدقيقة لاستخدام البناءات اليومية:

البناء اليومي. الجزء الأكثر أساسية من البناء اليومي هو الجزء "اليومي". كما يقول جيم مكارثي، تعامل مع البناء اليومي كنبض قلب للمشروع (مكارثي 1995). إذا لم يكن هناك دقات قلب، فإن المشروع قد مات. وبشكل أقل مجازًا، يصف مايكل كوسومانو وريتشارد سيلبي البناء اليومي كنبض متزامن لمشروع (كوسومانو و سيلبي 1995). يُسمح لشفرة المطورين المختلفين بالمزامنة قليلًا بين هذه النبضات، ولكن في كل مرة يكون هناك نبضة مزامنة، يجب أن تعود الشفرة إلى الانتظام. عندما تصر على إبقاء النبضات قريبة من بعضها البعض، فإنك تمنع المطورين من الخروج عن المزامنة تمامًا.

تقوم بعض المنظمات بعملية البناء كل أسبوع، بدلًا من كل يوم. تكمن المشكلة في ذلك في أنه إذا تم كسر البناء أسبوعًا، فقد يستغرق الأمر عدة أسابيع قبل البناء الجيد التالي. عندما يحدث ذلك، تفقد فعليًا جميع مزايا البناء المتكرر.

التحقق من البناءات المكسورة . لكي تنجح عملية البناء اليومية، يجب أن تعمل البرامج التي تم بناؤها. إذا لم يكن البرنامج قابلاً للاستخدام، فسيُعتبر البناء مكسورًا ويصبح إصلاحه أولوية قصوى.

يُحدد كل مشروع المعيار الخاص به لما يشكل "كسر البناء". يحتاج المعيار إلى تعيين مستوى جودة صارم بما فيه الكفاية لإبعاد العيوب المُعيقة للتقدم، ولكن مُتساهل بما فيه الكفاية لتجاهل العيوب الطفيفة، والتي يمكن أن تشل التقدم إذا أعطيت اهتمامًا لا مبرر له.

كحد أدنى، ينبغي على البناء "الجيد" ما يلي:

- ترجمة كافة الملفات والمكتبات والمكونات الأخرى بنجاح.
- ربط جميع الملفات والمكتبات والمكونات الأخرى بنجاح.
- عدم الاحتواء على أي أخطاء مُعيقة تمنع تشــغيل البرنامج أو تجعله خطرًا على العمل؛ وبعبارة أخرى، يجب أن يجتاز البناء الجيد اختبار الدخان.

اختبار الدخان اليومي. يجب أن يُنفذ اختبار الدخان على النظام بأكمله. ليس من الضروري أن يكون شاملا، ولكن يجب أن يكون قادرا على كشف المشاكل الكبيرة.

يجب أن يكون اختبار الدخان شاملا بما فيه الكفاية بحيث إذا اجتازه البناء، يمكنك الافتراض أنه مستقر بما يكفي لكي يتم اختباره كليًا.

إن للبناء اليومي قيمة قليلة دون اختبار الدخان. اختبار الدخان هو الحارس الذي يحمي من تدهور جودة المنتجات ومشاكل التكامل الزاحفة. وبدون ذلك، يصبح البناء اليومي مجرد ممارسة مضيعة للوقت في ضمان حصولك على ترجمة برمجية نظيفة كل يوم

الحفاظ على اختبار الدخان الحالي. يجب أن يتطور اختبار الدخان مع تطور النظام. في البداية، من المحتمل أن يختبر اختبار الدخان شيئًا بسيطًا، مثل ما إذا كان النظام يمكن أن يقول "مرحبًا بالعالم"[1]. مع تطور النظام، سيصبح اختبار الدخان أكثر دقة. قد يستغرق الاختبار الأول بضع ثوانٍ للتشغيل؛ ولكن مع نمو النظام، يمكن أن يزيد زمن اختبار الدخان إلى 10 دقائق أو ساعة أو أكثر. إذا لم يكن اختبار الدخان متجددًا، يمكن أن يصبح البناء اليومي تمرينًا في خداع الذات، حيث تخلق مجموعة ضئيلة من حالات الاختبار شعورًا خاطئًا بالثقة في جودة المنتج.

أتمتة البناء اليومي واختبار الدخان. رعاية وتغذية البناء يمكن أن تستغرق وقتًا طويلًا. تساعد أتمتة البناء واختبار الدخان على التأكد من بناء الشفرة وبدء اختبار الدخان. ليس من العملي بناء واختبار الدخان اليومي بدون التشغيل الأتوماتيكي.

إنشاء مجموعة بناء. في معظم المشاريع، يصبح التعامل مع البناء اليومي والحفاظ على اختبار الدخان حتى الآن مهمة كبيرة بما يكفي ليكون جزءًا صريحًا من وظيفة شخص واحد. أما في المشاريع الكبيرة، يمكن أن تصبح وظيفة بدوام كامل لأكثر من شخص. في الإصدار الأول من نظام التشغيل مايكروسوفت ويندوز إن تي `Microsoft Windows NT`، على سبيل المثال، كان هناك أربعة أشخاص بدوام كامل في مجموعة البناء (زخاري 1994).

إضافة مراجعات للبناء فقط عندما يكون من المنطقي القيام بذلك ... عادةً لا يكتب مطورو البرامج الفردية شفرة بشكل سريع بما فيه الكفاية لإضافة زيادات مفيدة للنظام على أساس يومي. يجب أن يعملوا على جزء من التعليمات البرمجية ومن ثم دمجها عندما يكون لديهم مجموعة من الشفرات في حالة متناسقة – عادةً يتم ذلك مرة كل بضعة أيام.

---

1 **هامش**: `hello world` "مرحبا، العالم!" عبارة عن برنامج بسيط يخرج أو يعرض للمستخدم عبارة "مرحبًا، بالعالم!". ويستخدم تقليديا لتقديم المبرمجين المبتدئين إلى لغة البرمجة. وكذلك لاختبار الصحة للتأكد من تثبيت لغة البرمجة بشكل صحيح، وفهم المشغل لكيفية استخدامها.

... لكن لا تنتظر وقتًا طويلاً لإضافة مجموعة المراجعات. حذار من التدقيق في التعليمات البرمجية بشكل غير منتظم. من الممكن أن يصبح أحد المطورين متورطًا في مجموعة من المراجعات التي يبدو أن كل ملف في النظام متورط فيها. وهذا يقوض قيمة البناء اليومي. وسيستمر باقي أعضاء الفريق في إدراك فائدة التكامل التزايدي، لكن ذلك المطور بالتحديد لن يدرك ذلك. إذا مضى على أحد المطورين أكثر من يومين دون التحقق من مجموعة من التغييرات، فخذ بعين الاعتبار أن عمل مطوّر البرامج مُعرض للخطر. وكما يشير كنت بيك، فإن التكامل المتكرر يجبرك أحيانًا على كسر بناء ميزة واحدة في حوادث عرضية مُتكررة. تُمثل هذه التكاليف غير المباشرة ثمناً مقبولاً مقابل خفض مخاطر التكامل، وتحسين وضوح الرؤية، وتحسين القدرة على الاختبار، والفوائد الأخرى للتكامل المتكرر (بيك 2000).

مُطالبة مطوري البرامج باختبار الدخان للشفرة الخاصة بهم قبل إضافتها إلى النظام. يحتاج المطورون إلى اختبار الشفرة الخاصة بهم قبل إضافتها إلى البنية. يمكن للمطور القيام بذلك عن طريق إنشاء بنية خاصة للنظام على جهازه الشخصي، ومن ثم يقوم المطور باختباره بشكل فردي. أو يمكن للمطوّر إصدار بنية خاصة إلى "صديق للاختبار"، وهو مختبِر يُركز على شفرة ذلك المطور. الهدف في كلتا الحالتين هو التأكد من أن الشفرة الجديدة تجتاز اختبار الدخان قبل أن يُسمح لها بالتأثير على أجزاء أخرى من النظام.

إنشاء منطقة احتجاز للشفرة المراد إضافتها إلى البنية. يعتمد جزء من نجاح عملية البناء اليومية على معرفة أي البناءات جيدة وأيها غير جيدة. عند اختبار الشفرة الخاصة بهم، يجب أن يكون المطورون قادرين على الاعتماد على نظام جيد معروف.

تقوم معظم المجموعات بحل هذه المشكلة عن طريق إنشاء منطقة احتجاز للشفرة، التي يعتقد المطورون أنها جاهزة للإضافة إلى البناء. تنتقل الشفرة الجديدة إلى منطقة الاحتجاز، ويتم بناء البنية الجديدة، وإذا كان البناء مقبولًا، يتم ترحيل الشفرة الجديدة إلى المصادر الرئيسة.

في المشاريع الصغيرة والمتوسطة الحجم، يمكن لنظام التحكم في الإصدارات أن يقوم بهذه الوظيفة. حيث يتحقق المطورين من الشفرة الجديدة في نظام التحكم بالإصدارات. ببساطة، يقوم المطورون الذين يريدون استخدام بنية جيدة معروفة بتعيين علامة تاريخ في ملف خيارات التحكم بالإصدار الذي يخبر النظام باسترداد الملفات استنادًا إلى تاريخ البنية الجيدة الأخيرة.

في المشاريع الكبيرة، أو المشاريع التي تستخدم برمجيات غير متطورة للتحكم في الإصدارات، يجب التعامل مع وظيفة منطقة الاحتجاز يدويًا. يرسل مؤلف مجموعة من الشفرات الجديدة بريد الإلكتروني إلى مجموعة البناء لإعلامهم بمكان العثور على الملفات الجديدة ليتم فحصها. أو تقوم المجموعة بإنشاء منطقة "فحص" على مخدم الملفات، حيث يضع المطورون إصدارات جديدة من ملفاتهم المصدر. بعد ذلك تحمل مجموعة البناء

المسؤولية عن التحقق من الشفرة الجديدة في التحكم في الإصدار بعد التحقق من أن الشفرة الجديدة لا تكسر البناء.

إنشاء عقوبة على كسر البناء. تقوم معظم المجموعات التي تستخدم البناءات اليومية بإنشاء عقوبة على كسر البناء. اجعل من الواضح منذ البداية أن الحفاظ على البناء الصحي يعد من أهم أولويات المشروع. يجب أن يكون البناء المكسور هو الاستثناء وليس القاعدة. الإصرار على أن يتوقف المطورين الذين قاموا بكسر البناء عن جميع الأعمال الأخرى حتى يتم إصلاحه. إذا تم كسر البناء كثيرا، فمن الصعب أن تأخذ مهمة عدم كسر البناء على محمل الجد.

يمكن أن تساعد عقوبة خفيفة في التأكيد على هذه الأولوية. بعض الجماعات تعطي مصاصة لكل أبله يكسر البناء. بعد ذلك، يضطر هذا المطور إلى أن يلصق المصاص على باب مكتبه حتى يصلح المشكلة. وهناك مجموعات أخرى لديها مطورين مذنبين يلبسون قرون الماعز أو يساهمون بمبلغ 5 دولارات في صندوق المعنويات.

تحدد بعض المشاريع عقوبة أكثر إحكاما. فقد اضطر مطوري مايكروسوفت في مشاريع رفيعة المستوى مثل حزمة البرامج المكتبية مايكروسوفت أوفيس وويندوز 2000 لارتداء جهاز استدعاء في المراحل الأخيرة من مشاريعهم. بحيث إذا قاموا بكسر البناء، يتم استدعاؤهم لإصلاحه حتى لو تم اكتشاف عيبهم في الساعة 3 صباحًا.

إصدار البناءات في الصباح. وجدت بعض المجموعات أنها تفضل البناء، واختبار الدخان في الصباح الباكر، وإصدار بناءات جديدة في الصباح وليس بعد الظهر. إن لاختبار الدخان وإصدار البناءات في الصباح له العديد من المزايا.

أولًا، في حالة إصدار بناء في الصباح، يمكن للمختبرين اختبار بناء جديد في ذلك اليوم. إذا كنت تصدر بشكل عام البناءات في فترة ما بعد الظهر، فإن المختبرين يشعرون أنهم مضطرون لبدء اختباراتهم الآلية قبل مغادرتهم لهذا اليوم. عندما يتأخر البناء، والذي غالبًا ما يتأخر، على المختبرين البقاء متأخرين لبدء اختباراتهم. نظرًا لأنه ليس خطأهم أن عليهم البقاء متأخرًا، تصبح عملية البناء مُحبطة.

عند إكمال البناء في الصباح، يكون لديك وصول أكثر موثوقية للمطورين عند وجود مشاكل في البناء. خلال اليوم، يكون المطورون في القاعة. أما خلال المساء، يمكن للمطورين أن يكونوا في أي مكان. حتى عندما يتم إعطاء المطوّرين أجهزة استدعاء، ليس من السهل دائمًا تحديد موقعهم.

قد يكون من الأفضل أن نبدأ اختبار الدخان في نهاية اليوم وأن ندعو الناس في منتصف الليل عندما تعثر على مشاكل، ولكن الأمر أصعب على الفريق، فهو يهدر الوقت، وفي النهاية تخسر أكثر مما تكسبه.

بناء واختبار الدخان حتى تحت الضغط. عندما يصبح ضغط الجدول الزمني مكثفًا، يمكن أن يبدو العمل المطلوب للحفاظ على البناء اليومي مثل النفقات الإضافية الباهظة. العكس هو الصحيح. تحت الضغط، يفقد المطورون بعضا من انضباطهم. إنهم يشعرون بالضغط لاتخاذ اختصارات البناء التي لن تتخذ في ظل ظروف أخرى أقل إرهاقًا. إنهم يراجعون ويختبرون الشفرة الخاصة بهم بشكل أقل حذرًا من المعتاد. تميل الشفرة نحو حالة الإنتروبيا بسرعة أكبر مما يحدث خلال الأوقات الأقل إجهادًا.

في مقابل هذا، تعمل البناءات اليومية على فرض الانضباط والحفاظ على مشاريع قِدر الضغط على الطريق الصحيح. حيث لا تزال الشفرة تميل نحو حالة من الإنتروبيا، لكن عملية البناء تجلب هذا الميل إلى الخلف في كل يوم. .

ما هي أنواع المشاريع التي يمكن أن تستخدم عملية البناء اليومية؟

يحتج بعض المطورين على أنه من غير العملي البناء كل يوم لأن مشاريعهم كبيرة جدًا. ولكن ما كان ربما أكثر مشاريع البرمجيات تعقيدًا في التاريخ الحديث استخدم البناء اليومي بنجاح. بحلول الوقت الذي تم إصداره، كان نظام مايكروسوفت ويندوز 2000 (Microsoft Windows 2000) يتألف من حوالي 50 مليون سطر من الشفرة موزعة عبر عشرات الآلاف من ملفات المصدر. استغرق *البناء* الكامل ما يصل إلى 19 ساعة على عدة أجهزة، ولكن فريق تطوير Windows 2000 لازال قادرا على البناء كل يوم. فبعيداً عن كونه مصدر إزعاج، عزا فريق Windows 2000 الكثير من نجاحه في هذا المشروع الضخم إلى بنائه اليومي. كلما كبر المشروع، أصبح التكامل التزايدي أكثر أهمية.

بيانات مثبتة

وجدت مراجعة لـ 104 مشروع في الولايات المتحدة، والهند، واليابان، وأوروبا أن 20- 25٪ فقط من المشاريع استخدمت البناءات اليومية إما في بداية أو وسط مشاريعهم (كوسومانو وآخرون 2003) ، لذا فإن هذا يمثل فرصة كبيرة للتحسين.

التكامل المستمر

اتخذّ بعض كتّاب البرامج بناءات يومية كنقطة انطلاق وأوصوا بالتكامل المستمر (بيك 2000). تستخدم معظم المراجع المنشورة للتكامل المستمر كلمة "مستمر" لتعني "على الأقل يوميًا" (بيك 2000)، وهذا ما أعتقد أنه معقول. لكنني أواجه أحيانًا أشخاصًا يأخذون كلمة "مستمر" حرفيًا. أنهم يسعون إلى تكامل كل تغيير مع أحدث بناء كل بضعة ساعات. بالنسبة لمعظم المشاريع، أعتقد أن التكامل المستمر بالمعنى الحرفي هو أمر جيد جدًا.

بيانات مثبتة

في وقت فراغي، أقوم بإجراء مناقشة مجموعة تتألف من كبار المسؤولين التنفيذيين الفنيين من شركات مثل امازون دوت كوم Amazon.com وبوينغ Boeing واكس بيديا Expedia ومايكروسوفت Microsoft ونوردستورم Nordstrom وغيرها من شركات منطقة سياتل. في استطلاع للرأي

من كبار المسؤولين التنفيذيين الفنيين، لم يظن أي منهم أن التكامل المستمر كان أفضل من التكامل اليومي. في المشروعات متوسطة الحجم والكبيرة، هناك قيمة في ترك الشفرة خارج المزامنة لفترات قصيرة. غالبًا ما يخرج المطوّرون عن المزامنة عند إجراء تغييرات على نطاق أوسع. يمكنهم بعد ذلك إعادة المزامنة بعد وقت قصير. تسمح البناءات اليومية لنقاط لقاء فريق المشروع التي تكون كافية في كثير من الأحيان. طالما أن الفريق يتزامن مع كل يوم فلا يحتاج إلى لقاء مستمر.

لائحة اختبار: التكامل[1]

استراتيجية التكامل

- هل تُحدد الاستراتيجية الترتيب الأمثل الذي يجب أن يتم فيه دمج الأنظمة الفرعية والصفوف والإجرائيات؟
- هل يتم تنسيق ترتيب الدمج مع ترتيب البناء بحيث تكون الصفوف جاهزة للتكامل في الوقت المناسب؟
- هل تؤدي الاستراتيجية إلى تشخيص العيوب بسهولة؟
- هل تُبقي الاستراتيجية السقالات في الحد الأدنى؟
- هل الاستراتيجية أفضل من النهج الأخرى؟
- هل تم تحديد الواجهات بين المكونات بشكل جيد؟ (تحديد واجهات ليست مهمة عملية التكامل، ولكن التحقق من أنها قد حُددت بشكل جيد هو مهمة عملية التكامل.)

البناء اليومي واختبار الدخان

- هل يتم بناء المشروع بشكل متكرر - مثالي، يومي - لدعم التكامل التزايدي؟
- هل يتم تشغيل اختبار الدخان مع كل بناء حتى تعرف ما إذا كان البناء يعمل؟
- هل قمت بأتمتة البناء واختبار الدخان؟
- هل يفحص المطورون شفرتهم بشكل متكرر— لا تتجاوز اليوم أو اليومين بين عمليات الفحص؟
- هل يبقى اختبار الدخان محدثًا مع الشفرة، مع التوسع وفقًا لتوسيع الشفرة؟
- هل كسر البناء أمر نادر الحدوث؟
- هل تقوم بالبناء واختبار الدخان للبرمجيات حتى عندما تكون تحت الضغط؟

[1] cc2e.com/2992

مصادر إضافية[1]

فيما يلي مصادر إضافية تتعلق بموضوعات هذا الفصل:

التكامل

Lakos, John. Large-Scale C++ Software Design. Boston, MA: Addison-Wesley, 1996.

لاكوس، جون. تصميم برمجيات سي ++ على نطاق واسع. بوسطن، أديسون ويسلي، 1996. يجادل لاكوس بأن "التصميم المادي" للنظام - التسلسل الهرمي للملفات والأدلة والمكتبات - يؤثر بشكل كبير على قدرة فريق التطوير على إنشاء برامج. إذا لم تهتم بالتصميم المادي، فستصبح أوقات البناء طويلة بما يكفي لتقويض التكامل المتكرر. تركز مناقشة لاكوس على لغة سي ++، لكن الأفكار المتعلقة بـ "التصميم المادي" تنطبق بنفس القدر على المشاريع بلغات أخرى.

Myers, Glenford J. The Art of Software Testing. New York, NY: John Wiley & Sons, 1979.

مايرز، جلينفورد. فن اختبار البرمجيات. نيويورك: جون وايلي وأولاده، 1979. يناقش كتاب الاختبار الكلاسيكي هذا التكامل كنشاط اختبار.

التزايد

McConnell, Steve. Rapid Development. Redmond, WA: Microsoft Press, 1996.

ماكونيل، ستيف. التطوير السريع. ريدموند، واشنطن: ميكروسوفت برس، 1996. يتناول الفصل 7، "تخطيط دورة الحياة"، الكثير من التفاصيل حول المقايضات المرتبطة بنماذج دورة حياة أكثر مرونة وأقل مرونة. تُناقش الفصول 20 و21 و35 و36 نماذج دورة حياة معينة، التي تدعم درجات مختلفة من التزايدية. يصف الفصل 19 "التصميم من أجل التغيير"، وهو نشاط أساسي ضروري لدعم نماذج التطوير التكرارية والتزايدية.

Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." Computer, May 1988: 61–72.

بويم، باري. "نموذج حلزوني لتطوير البرمجيات وتعزيزها". الحاسوب، مايو 1988: 61-72. في هذه الورقة، يصف بويم "نموذجه الحلزوني" لتطوير البرمجيات. يُقدم النموذج كنهج لإدارة المخاطر في مشروع تطوير البرمجيات، لذلك فإن الورقة البحثية تدور حول التطوير بشكل عام وليس عن التكامل على وجه التحديد. يُعتبر بويم أحد أهم الخبراء العالميين في مجال القضايا الكبيرة المتعلقة بتطوير البرمجيات، ويعكس وضوح تفسيراته جودة فهمه.

[1] cc2e.com/2999

Gilb, Tom. Principles of Software Engineering Management. Wokingham, England: Addison-Wesley, 1988.

جيلب، توم. مبادئ إدارة هندسة البرمجيات. ووكينغهام، إنجلترا: أديسون ويسلي، 1988. يحتوي الفصلان 7 و 15 على مناقشات دقيقة حول التسليم التطوري، وهو واحد من أول مقاربات التطوير التصاعدي.

Beck, Kent. Extreme Programming Explained: Embrace Change. Reading, MA: Addison-Wesley, 2000.

بيك، كنت. شرح البرمجة المتطرفة: قبول التغيير. ريدينغ: أديسون-ويسلي، 2000. يحتوي هذا الكتاب على عرض أكثر حداثة وأكثر إيجازًا للعديد من الأفكار في كتاب غيلب. أنا شخصيًا أفضل عمق التحليل المقدم في كتاب غيلب، لكن بعض القراء قد يجدون عرض بيك أكثر قابلية للوصول أو أكثر قابلية للتطبيق بشكل مباشر على نوع المشروع الذي يعملون عليه.

نقاط مفتاحية

- يؤثر تسلسل البناء ونهج التكامل على الترتيب الذي يتم به تصميم الصفوف وكتابة شفرتها واختبارها.
- يُقلل ترتيب التكامل المدروس بشكل جيد من جهد الاختبار ويخفف عمليات التصحيح.
- يأتي التكامل التزايدي في عدة أصناف، وما لم يكن المشروع بسيطا، فإن أي واحد منها أفضل من التكامل المرحلي.
- عادةً ما يكون نهج تكامل لأي مشروع محدد هو مزيج من نهج التكامل من الأعلى إلى الأسفل ومن الأسفل إلى الأعلى ومن التكامل الموجه نحو المخاطر وغيرها. التكامل على شكل حرف T وتكامل الساندويتش هما نهجان يعملان في كثير من الأحيان بشكل جيد.
- يمكن لعمليات البناء اليومية الحد من مشاكل التكامل، وتحسين معنويات المُطور، وتوفير معلومات مفيدة لإدارة المشاريع.

الفصل 30

أدوات البرمجة

المحتويات[1]

مواضيع ذات صلة

- أدوات التحكم في الإصدار: في القسم 28. 2
- أدوات *التصحيح*: القسم 23. 5
- أدوات دعم الاختبار: القسم 22. 5

تقلل أدوات البرمجة الحديثة من الوقت اللازم للبناء. يمكن أن يؤدي استخدام مجموعة أدوات متقدمة – والتآلف مع الأدوات المستخدمة - إلى زيادة الإنتاجية بنسبة 50 بالمئة أو أكثر (جونز، بويم) (Jones 2000؛ Boehm et al. 2000). كما يمكن لأدوات البرمجة أن تقلل من كمية الأعمال التفصيلية الشاقة التي تتطلبها البرمجة.



قد يكون الكلب هو أفضل صديق للرجل، في حين أن بعض الأدوات الجيدة هي أفضل الأصدقاء للمبرمج.

وكما اكتشف باري بويم منذ فترة طويلة، فإن 20 في المئة من الأدوات تميل إلى أن تكون مسؤولة عن 80 في المئة من استخدام الأداة (1987 ب). إذا كنت تفتقد إلى إحدى هذه الأدوات الأكثر فائدة، فأنت تفتقد شيئًا يمكنك استخدامه كثيرًا.

[1] cc2e.com/3084

يركز هذا الفصل على ناحيتين.

الأولى، إنها تغطي فقط أدوات البناء. إن طلب أدوات الإدارة والمواصفات وأدوات التطوير من النهاية إلى النهاية تقع خارج نطاق الكتاب. راجع قسم "الموارد الإضافية" في نهاية الفصل للحصول على مزيد من المعلومات حول الأدوات الخاصة بهذه الجوانب من تطوير البرامج.

الثانية، يغطي هذا الفصل أنواع الأدوات بدلاً من العلامات التجارية المحددة. هناك عدد قليل من الأدوات شائعة جدًا بحيث تتم مناقشتها بالاسم، ولكن الإصدارات والمنتجات والشركات تتغير بسرعة بحيث تصبح المعلومات حول معظمها قديمة حتى قبل أن يجف الحبر الموجود على هذه الصفحات.

يمكن للمبرمج العمل لسنوات عديدة دون اكتشاف بعض من الأدوات الأكثر قيمة المتاحة. مهمة هذا الفصل هي استكشاف الأدوات المتاحة. ومساعدتك في تحديد ما إذا كنت قد أغفلت أية أداة قد تكون مفيدة. إذا كنت خبيرًا في الأدوات، فلن تجد الكثير من المعلومات الجديدة في هذا الفصل. يمكنك تصفح الأجزاء السابقة من هذا الفصل، وقراءة القسم 30. 6 فيما يتعلق بـ "أداة أرض الخيال"، ثم انتقل إلى الفصل التالي.

30. 1 أدوات التصميم[1]

والهدف الأساسي من هذه الأدوات هو أتمتة عمليات توليد البرمجيات وتطوير التطبيقات. ويشمل ذلك أتمته جميع المراحل مثل: تحديد المتطلبات، والتحليل، والتصميم، وكتابة البرامج، والاختبار، والصيانة.

تتكون أدوات التصميم الحالية بشكل رئيسي من الأدوات الرسومية التي تنشئ مخططات التصميم.

يتم أحيانًا تضمين أدوات التصميم في أداة هندسة البرمجيات بمساعدة الحاسب (CASE) [2]مع وظائف أوسع؛ يعلن بعض البائعين عن أدوات تصميم مستقلة كأدوات CASE. تسمح لك أدوات التصميم الرسومية عمومًا بالتعبير عن التصميم من خلال الرموز الرسومية (طرق التدوين) الشائعة مثل: UML، أو مخططات الهيكلة البيانية، أو المخططات الهرمية، أو مخطط علاقة الكيانات، أو مخططات الصفوف. تدعم بعض أدوات التصميم الرسومية تدوينًا واحدًا فقط. في حين يدعم البعض الآخر عدّة تدوينات.

من ناحية، إن أدوات التصميم هذه هي مجرد حزم رسوم فاخرة. فباستخدام حزمة رسومات بسيطة أو قلم رصاص وورقة، يمكنك رسم كل شيء يمكن للأداة رسمه. ولكن الأدوات توفر لك إمكانات قيمة لا يمكن لحزمة رسومية بسيطة القيام بها. إذا كنت قد رسمت مخططًا فقاعيًا وحذفت فقاعة، فستقوم أداة التصميم الرسومية

[1] **إشارة مرجعية** للحصول على تفاصيل حول التصميم، راجع الفصول من 5 إلى 9.

[2] **هامش** "CASE" هندسة البرمجيات بمساعدة الحاسوب اختصار لـ: computer-aided software engineering

بإعادة ترتيب الفقاعات الأخرى تلقائيًا، بما في ذلك الأسهم المتصلة والفقاقيع ذات المستوى الأدنى المرتبطة بالفقاعة. كما تعتني الأداة بترتيب الأمور عند إضافة فقاعة أيضًا. يمكن لأداة التصميم أن تمكّنك من التنقل بين مستويات التجريد الأعلى والأدنى. ستقوم أداة التصميم بالتحقق من اتساق تصميمك، ويمكن لبعض الأدوات توليد شيفرة مباشرة من التصميم الخاص بك.

30. 2 أدوات الشفرة المصدرية

إن الأدوات المتوفرة للعمل مع الشفرة المصدرية أغنى وأكثر نضجا من الأدوات المتوفرة للعمل مع التصاميم.

التعديل (التحرير)

هذه المجموعة من الأدوات متعلقة بتحرير الشفرة المصدرية.

بيئات التطوير المتكاملة (IDEs)

بيانات مثبتة

يقدر بعض المبرمجين أنهم ينفقون ما يصل إلى 40 في المئة من وقتهم في تحرير الشفرة المصدرية (Parikh 1986، Ratliff 1987).

وإذا كان الأمر كذلك، فإن إنفاق بضعة دولارات إضافية للحصول على أفضل بيئة تطوير متكاملة ممكنة هو استثمار جيد. فبالإضافة إلى الوظائف الأساسية لمعالجة الكلمات، فإن بيئة تطوير جيدة تقدم هذه الميزات:

- التجميع والكشف عن الأخطاء من داخل المحرر.
- التكامل مع أدوات التحكم في الشفرة المصدرية، والبناء، والاختبار، وتصحيح الأخطاء.
- المعاينة المضغوطة أو المطويّة (أسماء الصفوف فقط أو التراكيب المنطقية من دون المحتويات، والمعروفة أيضًا باسم "الطي أو الثني")
- الانتقال السريع "القفز" إلى تعريفات الصفوف، والإجرائيات، والمتغيرات.
- الانتقال السريع إلى جميع الأماكن التي تم فيها استخدام الصف أو الإجرائية أو المتغير.
- تنسيق لغة محددة.
- مساعدة تفاعلية للغة التي يتم تحريرها
- حاضنة "مشبك" (begin-end) متجانسة
- قوالب لتراكيب اللغة الشائعة (على سبيل المثال يقوم المحرر بإكمال بنية حلقة for بعد كتابة المبرمج لكلمة for)
- المسافة البادئة الذكية (بما يضمن سهولة تغيير المسافة البادئة لكتلة من العبارات عند تغيير المنطق)

- عمليات إعادة التصنيع أو التحويل الأتوماتيكي للشفرة
- مسجلات "وحدات الماكرو" قابلة للبرمجة بلغة برمجة مألوفة
- إدراج سلاسل البحث بحيث لا تحتاج إلى إعادة كتابة السلاسل الشائعة الاستخدام
- التعابير النظامية في البحث والاستبدال
- البحث والاستبدال ضمن مجموعة من الملفات
- التحرير لملفات متعددة في وقت واحد
- مقارنات الفرق " diff" جنبا إلى جنب
- التراجع متعدد المستويات

بالنظر إلى بعض المحررات البدائية والتي لا تزال قيد الاستخدام، قد يفاجئك معرفة أن العديد من المحررات تضم كل هذه القدرات.

عمليات بحث واستبدال السلاسل النصية متعدد الملفات

إذا كان محررك لا يدعم البحث والاستبدال عبر ملفات متعددة، فإنه لا يزال بإمكانك العثور على أدوات داعمة للقيام بهذه المهمة. هذه الأدوات مفيدة للبحث عن كل تكرارات لاسم صف أو اسم إجرائية.

عندما تجد خطأ في شفرتك، يمكنك استخدام هذه الأدوات للتحقق من وجود أخطاء مشابهة في ملفات أخرى.

يمكنك البحث عن سلاسل "دقيقة" أو سلاسل متشابهة (مع تجاهل الاختلافات في الكتابة بالأحرف الكبيرة) أو التعابير النظامية. تعتبر التعابير النظامية قوية بشكل خاص لأنها تتيح لك البحث عن نماذج معقدة للسلسلة. إذا أردت العثور على كل المراجع التي تحتوي على أرقام سحرية (الأرقام من "0" إلى "9")، يمكنك البحث عن "["، متبوعًا بمسافات عددها صفر أو أكثر، متبوعًا برقم واحد أو أكثر، متبوعًا بمسافات عددها صفر أو أكثر، تليها "]". واحدة من أدوات البحث المتاحة على نطاق واسع تعرف باسم. "grep" سيبدو استعلام grep للأرقام السحرية كما يلي:

```
grep "\[ *[0–9] + * \]" * .cpp
```

يمكنك جعل معايير البحث أكثر تطورا لتحسين البحث.

من المفيد غالبًا أن تكون قادرًا على تغيير السلاسل النصية عبر ملفات متعددة. على سبيل المثال، إذا كنت تريد إعطاء متغير أو إجرائية أو ثابت أو متغير عام اسمًا أفضل، فقد يلزمك تغيير الاسم في عدة ملفات.

تجعل الأدوات المساعدة التي تسمح بتغييرات السلسلة عبر عدة ملفات ذلك سهلا، وهذا أمر جيد لأنه يجب أن يكون لديك أقل عدد ممكن من العوائق لإنشاء أسماء ممتازة للصفوف ولأسماء إجرائيات ولأسماء الثوابت.

تتضمن الأدوات الشائعة لمعالجة تغييرات سلسلة لملفات متعددة بيرل `Perl` و أوك AWK و سيد `sed`

أدوات المقارنة Diff

غالبًا ما يحتاج المبرمجون إلى مقارنة ملفين. إذا قمت بإجراء عدة محاولات لتصحيح خطأ وتحتاج إلى إزالة المحاولات غير الناجحة، فسيقوم مقارن الملفات بإجراء مقارنة بين الملفات الأصلية والمعدّلة وسرد الأسطر التي قمت بتغييرها.

إذا كنت تعمل على برنامج مع أشخاص آخرين وترغب في رؤية التغييرات التي أجروها منذ آخر مرة عملت فيها على الشفرة، فإن أداة مقارنة مثل `Diff` ستقوم بالمقارنة بين الإصدار الحالي وآخر إصدار من الشفرة عملت عليه وإظهار الاختلافات.

إذا اكتشفت عيبًا جديدًا لا تتذكر أنك واجهته في إصدار قديم من أحد البرامج، فبدلاً من رؤية طبيب أعصاب لعلاج فقدان الذاكرة، فيمكنك استخدام أداة مقارنة للمقارنة بين الإصدارات الحالية والقديمة من شفرة المصدر، وتحديد ما تم تغييره بالضبط، والعثور على مصدر المشكلة. وغالبًا ما يتم تضمين هذه الوظيفة في أدوات التحكم في المراجعة.

أدوات الدمج

يقوم أحد أنماط التحكم في المراجعة بقفل الملفات المصدرية بحيث يمكن لشخص واحد فقط تعديل ملف في كل مرة. بينما يسمح نمط آخر لعدة أشخاص بالعمل على الملفات في وقت واحد ويعالج دمج التغييرات في وقت فحص الهويات.

في وضع العمل هذا، تعتبر الأدوات التي تقوم بدمج التغييرات حساسة. وعادةً ما تقوم هذه الأدوات بإجراء عمليات دمج بسيطة تلقائيا وتستعلم من المستخدم عن عمليات الدمج التي تتعارض مع عمليات الدمج الأخرى أو تلك الأكثر تشابكا.

مجملات الشفرة المصدرية[1]

تعمل مجملات الشفرة المصدرية على تنظيم شفرتك المصدرية بحيث تبدو متناسقة. فهي تعلّم أسماء الصفوف والإجرائيات، وتوحّد نمط المسافة البادئة لديك، وتنسق التعليقات باستمرار، وتنفذ وظائف أخرى مماثلة.

[1] **إشارة مرجعية** للحصول على تفاصيل حول تخطيط البرنامج، راجع الفصل 31، "التنسيق والأسلوب".

يمكن لبعض المجملات وضع كل إجرائية في صفحة ويب منفصلة أو صفحة مطبوعة أو تنفيذ تنسيق أكثر دراماتيكية.

تتيح لك العديد من المجملات تخصيص الطريقة التي يتم بها تجميل الشفرة.

توجد فئتان على الأقل من أدوات تجميل الشفرة المصدرية. الفئة الأولى تأخذ الشفرة المصدرية كدخل وتنتج شفرة أفضل بكثير دون تغيير في شفرة المصدر الأصلية. والفئة الأخرى من الأدوات تغيّر شفرة المصدر نفسها - المسافة البادئة القياسية، وتنسيق قائمة المحددات، وما إلى ذلك.

هذه الإمكانية مفيدة عند العمل على كميات كبيرة من التعليمات البرمجية القديمة. يمكن للأداة القيام بالكثير من أعمال التنسيق المملة اللازمة لجعل الشفرة القديمة تتوافق مع اصطلاحات نمط كتابة الشفرة الخاص بك.

أدوات توثيق الواجهة

تقوم بعض الأدوات باستخراج توثيق مفصل عن واجهة المبرمج من ملفات الشفرة المصدرية. تستخدم الشفرة داخل الملف المصدر أدلة مثل حقول@tag لتحديد النص الذي يجب استخراجه. ثم تقوم أداة توثيق الواجهة باستخلاص النص الذي تم وضع علامة عليه وتقديمه بتنسيق جميل. إن Javadoc هي مثال شهير لهذا النوع من الأدوات.

القوالب

تساعدك القوالب في استغلال الفكرة البسيطة لتخفيف مهام لوحة المفاتيح التي غالبا ما تقوم بها وتريد أن تقوم بها بشكل متسق. لنفترض أنك تريد تعليق استهلالي قياسي في بداية إجرائيتك. يمكنك بناء هيكل الاستهلال مستخدما بناء جملة صحيح قواعديا مع تحديد الأماكن لكافة العناصر التي تريدها في الاستهلال القياسي. سيكون هذا الهيكل العظمي عبارة عن "قالب" تقوم بتخزينه في ملف أو "ماكرو" لوحة مفاتيح. وعند إنشاءك لإجرائية جديدة، يمكنك بسهولة إدراج القالب في ملف المصدر الخاص بك. يمكنك استخدام تقنية القالب لإعداد كيانات أكبر، مثل الصفوف والملفات، أو كيانات أصغر، مثل الحلقات.

إذا كنت تعمل في مشروع جماعي، فإن القوالب هي طريقة سهلة لتشجيع أساليب التشفير والتوثيق المتسقة. اجعل القوالب متاحة للفريق بأكمله في بداية المشروع، وسيستخدمها الفريق لأنها تجعل مهمته أسهل – حيث ستحصل على الاتساق كميّزة جانبية.

أدوات الإشارات المرجعية

تسرد أداة الإشارة المرجعية المتغيرات والإجرائيات وكل الأماكن التي تم استخدامهم فيها – بشكل نموذجي في صفحات الويب.

مولدات التصنيف الهرمي

ينتج مولد التصنيف الهرمي معلومات حول أشجار الوراثة.

وهذا يفيد أحيانًا في تصحيح الأخطاء، ولكن غالبًا ما يستخدم لتحليل بنية البرنامج أو تقسيم برنامج إلى حزم أو أنظمة فرعية. هذه الوظيفة متاحة أيضا في بعض بيئات التطوير المتكاملة IDEs.

تحليل جودة الشفرة

تقوم الأدوات الموجودة في هذا التصنيف بفحص الشفرة المصدرية الثابتة لتقييم جودتها.

مدققات انتقائية لبناء الجملة النحوي والدلالي

تدعم المدققات النحوية والدلالية المترجم الخاص بك عن طريق التحقق من الشفرة بشكل أكثر شمولاً من المترجم. قد يتحقق المترجم الخاص بك من أخطاء بناء الجملة النحوية فقط. قد يستخدم المدقق النحوي الانتقائي الفروق الدقيقة في اللغة للبحث عن أخطاء أكثر دقة - وهي أمور لا تكون خاطئة من وجهة نظر المترجم ولكن ربما لم تكن تنوي كتابتها. على سبيل المثال، في سي ++، العبارة

```
while ( i = 0 ) ...
```

إنها مقبولة تماما ولكن من المفترض أن تكون

```
while ( i = = 0 ) ...
```

السطر الأول صحيح من الناحية النحوية، ولكن التبديل بين = و == هو خطأ شائع وقد يكون السطر خاطئًا. لينت Lint هو مدقق نحوي ودلالي انتقائي يمكنك العثور عليه في العديد من البيئات سي / سي ++. يقوم لينت بتنبيهك عن المتغيرات غير المهيأة، والمتغيرات غير المستخدمة كليًا، والمتغيرات التي تم تعيين قيم لها ولم يتم استخدامها أبدا، ومحددات الإجرائية التي تم تمريرها من الإجرائية من دون اسناد قيمة لها، وعمليات المؤشر المشبوهة، والمقارنات المنطقية المشبوهة (كما في المثال المعروض للتو)، والشفرات التي لا يمكن الوصول إليها، والعديد من المشاكل الشائعة الأخرى. كما تقدم اللغات الأخرى أدوات مماثلة.

المخبرات عن المقاييس

بعض الأدوات تحلل شفرتك وتقدم تقريرًا عن جودتها[1]. على سبيل المثال، يمكنك الحصول على أدوات تعطي تقرير عن تعقيد كل إجرائية بحيث يمكنك استهداف الإجرائيات الأكثر تعقيدًا من أجل المراجعة الإضافية أو الاختبار أو إعادة التصميم. بعض الأدوات تقوم بإحصاء أسطر الشفرة وتصاريح البيانات والتعليقات والأسطر الفارغة في البرامج بالكامل أو في الإجرائيات الإفرادية. فهي تتبع العيوب وتربطها بالمبرمجين الذين قاموا بها، والتغييرات التي تُصححها، والمبرمجون الذين يقومون بالتصحيحات.

وهي تحصي التعديلات على البرنامج وتسجل الإجرائيات كثيرة التعديل. حيث وجد لأدوات تحليل التعقيد تأثير إيجابي بنسبة حوالي 20 في المئة على إنتاجية الصيانة (جونز 2000)

إعادة تصنيع الشفرة المصدرية

تساعد بعض الأدوات في تحويل شفرة المصدر من تنسيق إلى آخر.

معيدات التصنيع

يدعم برنامج إعادة التصنيع إعادة تصنيع الشفرات الشائعة إما بشكل مستقل أو بشكل مدمج في بيئة تطوير متكاملة (IDE). تتيح لك متصفحات إعادة التصنيع تغيير اسم الصف بسهولة عبر الشفرة بأكملها. فهي تسمح لك باستخراج الإجرائية ببساطة عن طريق تمييز الشفرة التي تريد تحويلها إلى إجرائية جديدة، وإدخال اسم الإجرائية الجديدة، وترتيب المحددات في قائمة محددات.

تجعل أدوات إعادة التصنيع تغييرات الشفرة البرمجية أسرع وأقل عرضة للخطأ. وهي متاحة لـ جافا وسمول توك وأصبحت متاحة للغات أخرى. لمزيد من المعلومات حول أدوات إعادة التصنيع، انظر الفصل 14، " أدوات إعادة التصنيع" في ***إعادة التصنيع*** *Refactoring* (فاولر 1999)

مُعيدات الهيكلة[2]

سيقوم معيد الهيكلة بتحويل صحن من شفرة السباغيتي مع تعليمات القفز gotos إلى طبق رئيسي من الشفرة المهيكلة بشكل أفضل ومن دون تعليمات gotos. أفاد كابرز جونز أنه في مجال صيانة البيئات، يمكن لأدوات إعادة هيكلة الشفرات أن يكون لها تأثير إيجابي بنسبة 25-30٪ على إنتاجية الصيانة (Jones 2000). يجب على القائم بإعادة الهيكلة أن يضع الكثير من الافتراضات عندما يحوّل الشفرة، وإذا كان المنطق مريعا في

---

[1] **إشارة مرجعية** لمزيد من المعلومات حول المقاييس، انظر القسم 28. 4 ، "القياس".

[2] **هامش** شفرة السباغيتي هو مصطلح يطلق على الشفرة المصدرية التي تملك بنية معقدة ومتشابكة من الأسطر البرمجية، والتي تنتج في الغالب عن استخدام أوامر الـ GOTO والاستثناءات، وتتجسد بكثرة في أعمال المبرمجين الأقل خبرة أو في المشاريع الكبيرة مفتوحة المصدر التي تخضع لعملية تحديث وتطوير مستمرة.

الأصل، فسيظل الأمر مريعا في النسخة المحوّلة. إذا كنت تجري عملية تحويل يدويًا، فعلى أي حال، يمكنك استخدام أداة إعادة هيكلة من أجل الحالة العامة وضبط الحالات الصعبة يدويا.

بديل عن ذلك، يمكنك تشغيل الشفرة من خلال أداة إعادة الهيكلة واستخدامها كعامل ملهم للتحويل اليدوي.

مترجمات الشفرة

تترجم بعض الأدوات الشفرة من لغة إلى أخرى. يفيد المترجم عندما يكون لديك شفرة كبيرة تنقلها إلى بيئة أخرى. الخطر في استخدام مترجم لغوي هو أنه إذا بدأت بشفرة سيئة، فببساطة سيترجم المترجم الشفرة السيئة إلى لغة غير مألوفة.

التحكم بالإصدار

إشارة مرجعية تم وصف هذه الأدوات وفوائدها في "تغييرات شفرة البرمجية" في القسم 28.2. يمكنك التعامل مع تزايد إصدارات البرمجية باستخدام أدوات التحكم في الإصدار من أجل

- التحكم في الشفرة المصدرية
- التحكم بالاعتمادية مثل تلك التي تقدمها الأداة المساعدة المرتبطة بـ يونكس UNIX
- ترقيم إصدارات توثيق المشروع
- ربط أدوات المشـــاريع مثل المتطلبات، والشـــفرة، وحالات الاختبار بحيث عندما يتغير متطلب ما، يمكنك العثور على الشفرة والاختبارات المتأثرة بذلك.

قواميس البيانات

إن قاموس البيانات هو عبارة عن قاعدة بيانات تصف جميع البيانات المهمة في مشروع. في كثير من الحالات، يركز قاموس البيانات بشكل أساسي على مخططات قاعدة البيانات. وفي المشاريع الكبيرة، يكون قاموس البيانات مفيدًا أيضًا لتتبع المئات أو الآلاف من تعريفات الصف. في مشاريع الفريق الكبيرة، من المفيد تجنب حدوث تضاربات في التسمية. قد يكون التضارب عبارة عن تضارب مباشر، حيث يستخدم فيه نفس الاسم مرتين، أو قد يكون تضاربا أكثر صعوبةً (أو فجوةً) يستخدم فيه أسماء مختلفة ليعني نفس الشيء أو يستخدم نفس الاسم ليعني أشياء مختلفة بشكل ماكر.

لكل عنصر بيانات (جدول قاعدة بيانات أو صف)، يحتوي قاموس البيانات على اسم العنصر ووصفه. وقد يحتوي القاموس أيضًا على ملاحظات حول كيفية استخدام العنصر.

30. 3 أدوات الشفرة القابلة للتنفيذ

إن أدوات التعامل مع الشفرة القابلة للتنفيذ كثيرة مثل أدوات التعامل مع الشفرة المصدرية.

إنشاء الشفرة

تساعد الأدوات الموصوفة في هذا القسم في إنشاء الشفرة.

المترجمات والرابطات

يحول المترجم الشفرة المصدرية إلى شفرة قابلة للتنفيذ. تتم كتابة معظم البرامج لتُترجم، على الرغم من أن بعضها تبقى مفسرة.

يربط الرابط القياسي ملف أو أكثر من ملفات الكائنات، والتي أنشأها المترجم من ملفاتك المصدرية، مع الشفرة القياسية المطلوبة لجعل البرنامج قابلا للتنفيذ. يمكن للرابطات عادةً ربط الملفات من لغات متعددة، مما يسمح لك باختيار اللغة الأكثر ملاءمة لكل جزء من برنامجك دون الحاجة إلى معالجة تفاصيل التكامل بنفسك.

يساعدك "رابط التراكب" overlay linker على وضع 10 أرطال في كيس بحجم خمسة أرطال عن طريق تطوير البرامج التي تُنفذ في ذاكرة أقل مقارنة بالمساحة الاجمالية التي تستهلكها. حيث ينشئ رابط التراكب ملفًا قابلاً للتنفيذ يحمّل جزءًا منه فقط إلى الذاكرة في وقت ما، تاركًا الباقي على القرص إلى حين الحاجة إليه.

أدوات البناء

الغرض من أداة البناء هو تقليل الوقت اللازم لإنشاء برنامج باستخدام الإصدارات الحالية من ملفات البرنامج المصدرية. لكل "ملف هدف" في مشروعك، تقوم بتحديد ملفات المصدر التي يعتمد عليها الملف الهدف وكيفية إنشائه. تزيل أدوات البناء أيضا الأخطاء المتعلقة بالمصادر التي تكون في حالات غير متناسقة؛ تضمن أداة البناء نقلها كلها إلى حالة متناسقة. تتضمن أدوات الإنشاء الشائعة أداة المساعدة المرتبطة بـ يونكس UNIX وأداة أنت ant المستخدمة في برامج جافا.

لنفترض أن لديك ملف هدف باسم userface.obj. في ملف make، تشير إلى أنه لعمل userface.obj، عليك ترجمة ملف userface.cpp. ويعني أيضًا أن userface.cpp يعتمد على userface.h و stdlib.h و project.h. ويعني مفهوم "يعتمد على" ببساطة أنه في حالة تغيير userface.h أو stdlib.h أو project.h، يجب أن يتم إعادة ترجمة userface.cpp.

عند بناء برنامجك، تتحقق أداة make من جميع التبعيات التي وصفتها وتحدد الملفات التي تحتاج إلى إعادة ترجمة. إذا كانت خمسة من ملفات المصدر الـ 250 الخاصة بك تعتمد على تعريفات البيانات في userface.h وتتغير، فإن make تلقائيا يعيد ترجمة الملفات الخمسة المعتمدة عليها. ولا يعيد ترجمة الملفات الـ 245 التي لا تعتمد على userface.h. استخدام ميك make أو أنت ant يتفوق على بدائل إعادة ترجمة جميع الملفات 250 أو إعادة ترجمة كل ملف يدويا، ونسيان واحد، والحصول على أخطاء خروج عن التزامن غريبة. وبشكل عام، فإن أدوات البناء مثل make أو ant تحسن بشكل كبير زمن وموثوقية متوسط دورة الترجمة – الارتباط – التشغيل.

وجدت بعض المجموعات بدائل مهمة لأدوات التحقق من التبعية مثل ميك make. على سبيل المثال، وجدت مجموعة مايكروسوفت وورد ببساطة أن إعادة بناء كافة ملفات المصدر كانت أسرع من إجراء فحص شامل للاعتمادية مع جعل ملفات المصدر نفسها محسّنة (محتويات ملف الرأس وما إلى ذلك).

باستخدام هذا الأسلوب، يمكن لجهاز متوسط لمطوِّر على مشروع وورد إعادة بناء الملفات القابلة للتنفيذ بالكامل لبرنامج وورد - عدة ملايين من أسطر الشفرة - في حوالي 13 دقيقة.

مكتبات الشفرة

إن الطريقة الجيدة لكتابة شفرة عالية الجودة في فترة زمنية قصيرة لا تتمثل في كتابة كل شيء ولكن بإيجاد إصدار مفتوح المصدر أو شرائه بدلاً من ذلك. يمكنك العثور على مكتبات عالية الجودة على الأقل في هذه المجالات:

- الصفوف الحاوية Container classes
- خدمات مناقلات بطاقات الائتمان (خدمات التجارة الإلكترونية)
- أدوات التطوير متعددة المنصـــات. قد تكتب شـــفرة تنفذ في مايكروســـوفت ويندوز وآبل ماكنتوش Apple Macintosh ونظام ويندوز اكس Window X فقط عن طريق إعادة الترجمة إلى كل بيئة.
- أدوات ضغط البيانات
- أنواع البيانات والخوارزميات
- عمليات قواعد البيانات وأدوات معالجة ملفات البيانات
- ادوات الرسوم البيانية، ورسم المنحنيات البيانية ورسم المخططات
- أدوات التصوير
- مديرات التراخيص

- العمليات الرياضية
- أدوات الشبكات واتصالات الإنترنت
- مولدات التقارير وبنّائات استعلامات التقارير
- أدوات الأمان والتشفير
- أدوات الجداول الالكترونية والجداول
- أدوات النص والتهجئة
- أدوات الصوت والهاتف والفاكس

مساعدات توليد الشفرة

إذا لم تتمكن من العثور على الشفرة التي تريدها، فما رأيك في جعل شخص آخر يكتبها بدلاً من ذلك؟ لا يجب عليك ارتداء السترة الصفراء ذات المربعات المنقوشة وتبدأ بثرثرة رجل مبيعات السيارات لإقناع شخص آخر بكتابة شفرتك.

يمكنك العثور على أدوات تكتب لك الشفرة، وغالبًا ما يتم دمج هذه الأدوات في بيئة التطوير المتكاملة IDE.

تميل أدوات توليد الشفرة إلى التركيز على تطبيقات قواعد البيانات، ولكن هذا يشمل الكثير من التطبيقات. مولدات الشفرة الشائعة تكتب الشفرات لقواعد البيانات، وواجهات المستخدم، والمترجمات. نادرًا ما تكون الشفرة المولَّدة بجودة الشفرة التي يكتبها مبرمج حقيقي "بشري"، ولكن العديد من التطبيقات لا تتطلب شفرة يدوية. فمن المفيد أكثر لدى بعض المستخدمين الحصول على 10 تطبيقات تعمل "شغالة" بدلاً من الحصول على تطبيق واحد يعمل بشكل مثالي.

إن مولدات الشفرة مفيدة أيضًا لعمل نماذج أوليه من شفرة الإنتاج "النهائية". باستخدام مُولِّد شفرة، قد تتمكن من إنشاء نموذج أولي في غضون بضعة ساعات، يوضح الجوانب المفتاحية لواجهة المستخدم أو قد تتمكن من تجربة مناهج تصميم مختلفة. الأمر الي قد يستغرق منك عدة أسابيع في البرمجة اليدوية للكثير من الوظائف. إذا كنت تجرب فقط، فلماذا لا تفعل ذلك بأرخص طريقة ممكنة؟

يتمثل العيب الشائع لمولدات الشفرة في أنها تميل إلى إنشاء شفرة غير قابلة للقراءة تقريبًا. فإذا احتجت في وقت ما إلى شفرة كهذه، فيمكن أن تندم على عدم كتابتها يدويًا من المرة الأولى.

الإعداد والتنصيب

يوفر العديد من الموردين أدوات تدعم إنشاء برامج الإعداد "Setup". تدعم هذه الأدوات عادة إنشاء الأقراص أو الأقراص المضغوطة CD، أو أقراص الفيديو الرقمية DVD، أو التثبيت عبر الويب. حيث تتحقق من وجود ملفات المكاتب الشائعة بالفعل في الجهاز المطلوب التنصيب عليه، وتفحص الإصدار، وما إلى ذلك.

المعالجات التمهيدية Preprocessors [1]

تعتبر المعالجات التمهيدية ودوال مسجلات "ماكرو" للمعالج التمهيدي مفيدة لتصحيح الأخطاء لأنها تجعل من السهل التبديل بين شفرة التطوير وشفرة الإنتاج. إذا كنت تريد التحقق من تجزئة الذاكرة في بداية كل إجرائية أثناء التطوير، فيمكنك استخدام ماكرو ليُنفذ في بداية كل إجرائية. وقد لا ترغب في الإبقاء على عملية التحقق هذه في شفرة الإنتاج، وبالتالي يمكنك إعادة تعريف الماكرو من أجل شفرة الإنتاج بحيث لا يقوم بتوليد أية شفرة على الإطلاق. ولأسباب مشابهة، تعتبر وحدات ماكرو المعالج التمهيدي جيدة لكتابة الشفرات المراد ترجمتها في بيئات متعددة - على سبيل المثال، في كل من نظامي التشغيل ويندوز ولينوكس.

إذا كنت تستخدم لغة ذات بنى تحكم بدائية، مثل المجمّع assembler، يمكنك كتابة معالج مسبق لتدفق التحكم لمحاكاة بنى التركيب لـ if-then-else وحلقات while في لغتك.

وإذا لم تكن لغتك تحتوي على معالج مسبق، فيمكنك استخدام معالج أولي مستقل كجزء من عملية البناء[2]. أحد المعالجات التمهيدية المتاح بسهولة ويسر هوM4 ، متوفر من خلال الرابط www.gnu.org/software/m4/.

التصحيح Debugging[3]

تساعد هذه الأدوات في تصحيح الأخطاء:

- رسائل تحذير المترجم
- اختبار السقالات
- أدوات الفرق Diff (لمقارنة الإصدارات المختلفة لملفات الشفرة المصدرية)
- موصفات التنفيذ
- مراقبات التتبع
- المصححات التفاعلية - أدوات اختبار كل من المكونات المادية والبرمجية

[1] إشارة مرجعية للحصول على تفاصيل حول نقل مساعدات التصحيح داخل وخارج الشفرة، راجع "خطة لإزالة مساعدات التصحيح" في القسم 8. 6.

[2] cc2e.com/3091

[3] إشارة مرجعية هذه الأدوات وفوائدها موصوفة في القسم 23. 5، "أدوات التصحيح – الواضحة والواضحة تقريبا."

- أدوات الاختبار المناقشة تاليًا، متعلقة بأدوات التصحيح.

الاختبار [1]

يمكن أن تساعدك هذه الميزات والأدوات على إجراء اختبار فعال:

- منصات الاختبار التلقائي مثل جي يونيت JUnit، إن يونت NUnit، سي بي بي يونت CppUnit، وغيرها
- مولدات الاختبار التلقائي
- خدمات إعادة التشغيل playback وتسجيل حالات الاختبار
- مراقبات التغطية (محللات المنطق وموصِّفات التنفيذ)
- المصححات الرمزية
- اضـــطرابات النظام (مالئات الذاكرة، هزازات الذاكرة memory shakers، عاطبات الذاكرة الانتقائية، مدققات الوصول إلى الذاكرة)
- أدوات Diff (لمقارنة ملفات البيانات والناتج الملتقط وصور الشاشة)
- السقالات
- أدوات حقن العيوب
- برمجيات تعقب العيوب

ضبط الشفرة

يمكن أن تساعدك هذه الأدوات في ضبط شفرتك بشكل جيد.

موصفات التنفيذ

يراقب موصف التنفيذ شفرتك البرمجية أثناء تشغيلها ويخبرك بعدد المرات التي يتم فيها تنفيذ كل تعليمة أو مقدار الوقت الذي يقضيه البرنامج في كل تعليمة أو مسار التنفيذ. إن توصيف الشفرة أثناء تشغيلها يشبه قيام الطبيب بالضغط بسماعته على صدرك وطلبه منك بالسعال. فهي تعطيك نظرة ثاقبة عن كيفية عمل برنامجك، وأين توجد النقاط الفعالة، وأين يجب عليك تركيز جهود ضبط الشفرة.

لوائح المجمع والمفكك

[1] **إشارة مرجعية** تم وصف هذه الأدوات وفوائدها في القسم 22. 5، "أدوات دعم الاختبار".

قد ترغب في يوم ما في الاطلاع على شفرة المجمّع المُنشَئة بلغتك عالية المستوى. تولّد بعض مترجمات اللغات عالية المستوى لوائح المجمّع. والبعض الآخر لا. ويتوجب عليك استخدام وحدة فك تجميع لإعادة إنشاء المجمع من شفرة الآلة التي ولّدها المترجم. وبالنظر إلى شفرة المجمع التي ولدها المترجم الخاص بك، يظهر لك مدى كفاءة مترجمك في ترجمة شفرة اللغة عالية المستوى إلى شفرة الآلة. من الممكن أن يظهر لك سبب تشغيل الشفرات عالية المستوى- والتي تبدو سريعة- ببطء. في الفصل 26، "تقنيات ضبط الشفرة،" العديد من النتائج القياسية غير متوقعة. أثناء قياس تلك الشفرات، أشرت مرارًا إلى لوائح المجمعات لفهم أفضل للنتائج التي لم تكن منطقية في اللغة عالية المستوى.

إذا لم تكن مرتاحًا بلغة التجميع وتريد توضيح، لن تجد أفضل من مقارنة كل تعليمة كتبتها بلغة عالية المستوى بتعليمات المجمّع المولّدة بواسطة المترجم. أول تعرُّض للمجمّع في كثير من الأحيان هو فقدان "البراءة". فعندما ترى مقدار الشفرة التي ينشئها المترجم - ما هو أكثر بكثير مما تحتاج إليه - لن تنظر أبدًا إلى مترجمك بنفس الطريقة مرة أخرى.

على العكس، في بعض البيئات، يجب على المترجم توليد شفرة معقدة للغاية. يمكن لدراسة خرج المترجم أن تعزز التقدير لمدى الحاجة إلى العمل في البرنامج بلغة ذات مستوى أخفض.

30. 4 البيئات الموجهة بالأدوات

أثبتت بعض البيئات أنها مناسبة بشكل أفضل للبرمجة الموجهة بالأدوات أكثر من غيرها.

تشتهر بيئة يونكس بمجموعة أدواتها الصغيرة ذات الأسماء المسلية والتي تعمل معا بشكل جيد: `grep` و`diff` و`sort` و`make` و`crypt` و`tar` و`lint` و`ctags` و`sed` و`awk` و`vi` وغيرها. تشتمل لغات سي و سي ++، المقترنة بشكل وثيق مع يونكس على نفس الفلسفة، حيث تتكون مكتبة سي ++ القياسية من دوال صغيرة يمكن أن تدمج بسهولة في دوال أكبر لأنها تعمل بشكل جيد معًا.

يعمل بعض المبرمجين بشكل مثمر في نظام يونكس والذي يأخذونه معهم[1]. حيث يستخدمون أدوات تشابه يونكس بالعمل لدعم عادات يونكس الخاصة بهم في بيئة الويندوز والبيئات الأخرى.

أحد الثمرات لنجاح نموذج يونكس هو توافر الأدوات التي تضع حلّة يونكس على الأجهزة الأخرى. على سبيل المثال، يوفر `cygwin` أدوات مكافئة ليونكس التي تعمل على نظام ويندوز (`www.cygwin.com`).

[1] cc2e.com/3026

يحتوي فن برمجة اليونكس (The Art of Unix Programming) (2004) لـ إريك رايموند على مناقشة عميقة لثقافة البرمجة في يونكس.

30. 5 بناء أدواتك البرمجية الخاصة

لنفترض أنك أُعطيت خمس ساعات لإنجاز مهمة ولديك الخيار:

- القيام بالمهمة بشكل مريح في خمس ساعات، أو
- قضاء أربع ساعات و45 دقيقة في بناء أداة للقيام بهذه المهمة، ومن ثم سيكون لديك الأداة التي تؤدي المهمة خلال 15 دقيقة.

سيختار معظم المبرمجين الجيدين الخيار الأول مرة واحدة من مليون والخيار الثاني في كل الحالات الأخرى. أدوات البناء هي جزء من أساس ونسيج البرمجة. تحتوي جميع المنظمات الكبيرة تقريبًا (المنظمات التي تضم أكثر من 1000 مبرمج) على أدوات داخلية ومجموعات دعم. لدى العديد منها متطلبات ملكية وأدوات تصميم تتفوق على تلك الموجودة في السوق (جونز 2000).

يمكنك كتابة العديد من الأدوات الموضحة في هذا الفصل. قد لا يكون القيام بذلك فعالا من حيث التكلفة، ولكن لا توجد أي عوائق تقنية هائلة تمنع القيام بذلك.

أدوات مشاريع محددة

تحتاج معظم المشاريع المتوسطة والكبيرة إلى أدوات خاصة فريدة للمشروع. على سبيل المثال، قد تحتاج إلى أدوات لتوليد أنواع خاصة من بيانات الاختبار، أو للتحقق من جودة ملفات البيانات، أو محاكاة الأجهزة غير المتوفرة بعد. فيما يلي بعض الأمثلة عن دعم أداة خاصة بالمشروع:

- كان فريق الطيران هو المسؤول عن تطوير برمجيات الطيران للتحكم بجهاز استشعار "حساس" الأشعة تحت الحمراء وتحليل بياناته. للتحقق من أداء البرمجية، وثّق مسجل بيانات الطيران إجراءات برمجية الطيران. وكتب المهندسون أدوات تحليل بيانات مخصصة لتحليل أداء أنظمة الطيران. كانوا يستخدمون الأدوات المخصصة للتحقق من الأنظمة الأساسية بعد كل رحلة.
- خططت مايكروسوفت لتضمين تقنية خطوط جديدة في إصدار من بيئتها الرسومية لـويندوز. وبما أن كل من ملفات بيانات الخط وبرمجية عرض الخطوط كانت جديدة، فيمكن أن تنشأ أخطاء من البيانات أو من البرمجية. كتب مطورو مايكروسوفت العديد من الأدوات المخصصة للتحقق من وجود أخطاء في ملفات البيانات، مما أدى إلى تحسين قدرتهم على التمييز بين أخطاء بيانات الخطوط وأخطاء البرمجية.

- طورت شــركة تأمين نظامًا طموحًا لحســاب معدلات النمو لديها. ولأن النظام كان معقدًا وكانت الدقة ضرورية، وكانت المئات من المعدلات المحسوبة محتاجة للتحقق منها بعناية، على الرغم من أن عملية حساب معدل واحدة تستغرق عدة دقائق. كتبت الشركة أداة برمجية منفصلة لحساب معدلات عملية واحدة في كل مرة. باستخدام الأداة، يمكن أن تحسب الشركة معدلا واحدًا في بضع ثوانٍ وتتحقق من المعدلات من البرنامج الرئيسي في جزء صغير من الوقت الذي كانت ستستغرقه للتحقق من معدلات البرنامج الرئيسي يدويًا.

ينبغي أن يكون جزء من التخطيط لمشروع ما هو التفكير في الأدوات التي قد تكون مطلوبة وتخصيص وقت لبنائها.

التدوينات "البرامج النصية" Scripts

السكريبت هو أداة تقوم بأتمتة الأعمال الروتينية المتكررة. في بعض الأنظمة، تُسمى السكريبتات بـ الملفات الدفعية أو وحدات الماكرو. يمكن أن تكون البرامج النصية بسيطة أو معقدة، والأكثر فائدة بينها هو الأسهل في الكتابة. على سبيل المثال، أحتفظ بدفتر يوميات، ولحماية خصوصيتي، أقوم بتشفيرها إلا عندما أكتبها. للتأكد من أن تشفيرها وفك تشفيرها يتم دوما بشكل صحيح، لدي سكريبت يفك تشفير دفتر اليومية الخاص بي، وينفذ معالج الكلمات، ثم يقوم بتشفير دفتر اليومية. يبدو النص مثل هذا:

```
crypto c: \ word \ journal. *1 ٪ / d / Es / s
word c: \ word \ journal.doc
crypto c: \ word \ journal. *1 ٪ / Es / s
```

يمثل 1٪ حقل كلمة المرور الخاصة بي والتي لم يتم تضمينها في السكريبت لأسباب واضحة. يوفر لي السكريبت عمل كتابة (وأخطاء كتابة) جميع المحددات ويضمن أنني دائماً أقوم بأداء كل العمليات وأقوم بتنفيذها بالترتيب الصحيح.

إذا وجدت نفسك تكتب شيئًا أطول من خمس محارف أكثر من بضع مرات في اليوم، فهو مرشح جيد لسكريبت أو ملف دفعي. تتضمن الأمثلة تسلسلات الترجمة / الارتباط، وتعليمات النسخ الاحتياطي، وأي أمر يحتوي على الكثير من المحددات.

30. 6 أداة أرض الخيال[1]

لعقود من الزمان، وعد بائعو الأدوات وخبراء الصناعة بأن الأدوات اللازمة لإقصاء البرمجة أصبحت في الافق. كانت أول أداة، وربما الأكثر إثارة للسخرية، لتلقي هذا اللقب، هي فورتران. تم تصميم لغة فورتران أو "لغة الترجمة المعيارية" بحيث يمكن للعلماء والمهندسين ببساطة كتابة الصيغ، وبالتالي يفترض أن تلغي الحاجة إلى المبرمجين.

نجحت فورتران في تمكين العلماء والمهندسين من كتابة البرامج، لكن من موقعنا اليوم، تبدو فورتران لغة برمجة منخفضة نسبيًا. بالكاد استبعدت الحاجة للمبرمجين، الخبرة التي اكتسبتها هذه الصناعة من فورتران هي مؤشر التقدم في صناعة البرمجيات ككل..

تطوّر صناعة البرمجيات باستمرار أدوات جديدة تقلل أو تلغي بعضا من أكثر جوانب البرمجة مَللا: كتفاصيل تصميم عبارات المصدر البرمجية، والخطوات اللازمة للتحرير edit، والترجمة compile، والربط، وتشغيل برنامج؛ والعمل المطلوب للعثور على الأقواس غير المتطابقة؛ وعدد الخطوات اللازمة لإنشاء صناديق رسائل قياسية؛ وما إلى ذلك. ومع بدء كل من هذه الأدوات الجديدة في إثبات المكاسب الإضافية في الإنتاجية، قدّر الخبراء أن هذه المكاسب لا نهائية، بافتراض أن المكاسب ستؤدي في النهاية إلى "القضاء على الحاجة إلى البرمجة". ولكن ما يحدث في الواقع هو أن كل ابتكار برمجي جديد يصل مع بعض العيوب. ومع مرور الوقت، تُزال العيوب وتتحقق كامل إمكانيات الابتكار. ومع ذلك، بمجرد إنجاز مفهوم الأداة الأساسي، تُنجز المزيد من المكاسب عن طريق إزالة الصعوبات العرضية التي تنشأ كآثار جانبية لإنشاء أداة جديدة. لا يؤدي القضاء على هذه الصعوبات العرضية إلى زيادة الإنتاجية في حد ذاتها؛ إنها ببساطة تقضي على الـ "خطوة إلى الوراء" من المعادلة النموذجية "خطوتين إلى الأمام، خطوة إلى الوراء".

على مدى العقود العديدة الماضية، شهد المبرمجون العديد من الأدوات التي كان من المفترض أن تلغي البرمجة. في البداية كانت لغات الجيل الثالث. ثم لغات الجيل الرابع. ثم جاءت البرمجة التلقائية. ثم أدوات CASE (هندسة البرمجيات بمساعدة الحاسوب). وبعدها البرمجة المرئية. وقد نجحت كل من هذه التطويرات في زيادة التحسينات القيّمة والتدريجية على برمجة الحاسوب، وبشكل جماعي جعلت من البرمجة شيئاً لا يمكن التعرف عليه من قبل أي شخص تعلم البرمجة قبل هذه التطويرات. لكن أيا من هذه الابتكارات لم تنجح في إلغاء البرمجة.

---

[1] **إشارة مرجعية** يعتمد توفر الأداة جزئيًا على نضج البيئة التقنية. لمزيد من المعلومات حول هذا، راجع القسم 4. 3 ، "موقعك على الموجة التكنولوجية".

والسبب في هذه الديناميكية هو أن البرمجة في جوهرها صعبة أساسا - حتى مع الدعم من خلال الأدوات الجيدة[1]. بغض النظر عن الأدوات المتاحة، سيكون على المبرمجين أن يتصارعوا مع العالم الحقيقي الفوضوي. سيكون علينا التفكير بصرامة حول التسلسلات، والتبعيات، والاستثناءات؛ وسنضطر للتعامل مع المستخدمين النهائيين الذين لا يمكن إرضاء مزاجهم. سيتعين علينا دائمًا أن نتصارع مع واجهات غير معرفة بدقة للمكونات المادية والبرمجية الأخرى، وسنضطر إلى حساب مبادئ المشاريع التجارية والقوانين وغيرها من مصادر التعقيد التي تنشأ من خارج عالم برمجة الحاسوب.

سنكون بحاجة دائماً إلى أشخاص يستطيعون سد الفجوة بين مشكلة العالم الحقيقي التي يتوجب حلها والحاسوب الذي من المفترض أن يحل المشكلة. سيطلق على هؤلاء الأشخاص اسم المبرمجين بغض النظر عما إذا كنا نتعامل مع سجلات الجهاز في المجمّع أو صناديق الحوار في مايكروسوفت فيجوال بيسك. طالما لدينا حواسيب، سنحتاج إلى أشخاص يخبرون أجهزة الحاسب بما يجب فعله، وسيطلق على هذا النشاط اسم البرمجة.

عندما تسمع زعم بائع أداة يقول "هذه الأداة الجديدة ستقضي على برمجة الحاسوب"، امض! أو على الأقل ابتسم بينك وبين نفسك على تفاؤل البائع الساذج.

مصادر إضافية

ألقِ نظرة على هذه الموارد الإضافية لمزيد من المعلومات عن أدوات البرمجة[2]:

يعتبر موقع `www.sdmagazine.com/jolts` مجلة تطوير البرمجيات السنوية لجائزة الإنتاجية بمثابة مصدر جيد للمعلومات حول أفضل الأدوات الحالية[3].

يقدم القسم الثالث من كتاب "مبرمج واقعي" لـ هانت، وأندرو، وديفيد توماس. ام أي: أديسون ويسلي 2000
.

`Hunt, Andrew and David Thomas.` *`The Pragmatic Programmer`*`. Boston, MA: AddisonWesley, 2000`

مناقشة عميقة لأدوات البرمجة، بما في ذلك المحررات ومولدات الشفرة، والمصححات، والتحكم في الشفرة المصدرية، وأدوات أخرى ذات الصلة.

[1] **إشارة مرجعية** أسباب صعوبة البرمجة موصوفة في "الصعوبات العرضية والضرورية" في القسم 2.5.

[2] cc2e.com/3098

[3] cc2e.com/3005

فون-نيكولز ، ستيفن. "بناء برامج أفضل باستخدام أدوات أفضل[1] "، IEEE Computer ، أيلول 2003 ، الصفحتان 12-14. تستعرض هذه المقالة مبادرات الأدوات التي تقودها أبحاث أي بي أم، ومايكروسوفت، وصن. IBM و Sun.

Vaughn-Nichols, Steven. "Building Better Software with Better Tools," *IEEE Computer*, September 2003, pp. 12–14. This article surveys tool initiatives led by IBM, Microsoft Research, and Sun Research.

Glass, Robert L." "تعارض البرمجيات: مقالات حول فن وعلوم هندسة البرمجيات
Glass, Robert L. *Software Conflict: Essays on the Art and Science of Software Engineering*. Englewood Cliffs, NJ: Yourdon Press, 1991.

يوفر الفصل المعنون "موصى به: الحد الأدنى من مجموعة أدوات البرمجيات القياسية" وجهة نظر مدروسة لأفضل الأدوات. provides a thoughtful counterpoint to the more-tools-is-better view.

يناقش Glass تحديد الحد الأدنى من الأدوات التي يجب أن تكون متاحة لجميع المطورين ويقترح عدّة البدء.

جونز ، كابير. تقدير تكاليف البرمجيات. New York، NY: McGraw-Hill، 1998.

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998.

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: AddisonWesley, 2000

يخصص كل من كتب جونز و بويم أقسامًا لتأثير استخدام الأداة على الإنتاجية

قائمة التحقق: أدوات البرمجة[2]

- هل لديك بيئة تطوير متكاملة فعالة؟
- هل تدعم بيئة التطوير المتكاملة لديك التكامل مع التحكم بالشـــفرة المصـــدرية؛ والبناء والاختبار، وأدوات التصحيح. والدوال المفيدة الأخرى؟
- هل لديك أدوات تقوم بأتمتة عمليات إعادة التصنيع الشائعة
- هل تســـتخدم التحكم في الإصـــدار لإدارة الشـــفرة المصـــدرية، والمحتوى، والمتطلبات، والتصميمات، وخطط المشروع، وغيرها من الأعمال الفنية الخاصة بالمشروع؟

[1] cc2e.com/3012
[2] cc2e.com/3019

- إذا كنت تعمل في مشروع كبير جدًا، فهل تستخدم قاموس بيانات أو مستودعًا مركزيًا آخر يحتوي على أوصاف موثوقة لكل صف مستخدم في النظام؟
- هل فكرت في مكتبات الشفرة كبدائل لكتابة الشفرات المخصصة، حيثما يكون ذلك متاحا؟
- هل تستخدم مصحح تفاعلي؟
- هل تستخدم Make أو برمجية أخرى للتحكم في التبعية لإنشاء برامج بكفاءة وموثوقية؟
- هل تتضمن بيئة الاختبار لديك منصة اختبار مؤتمتة، ومولدات اختبار آلية، ومراقبات تغطية coverage monitors، ومشوشات النظام system perturbers، وأدوات الفرق، وتتبع خلل البرمجيات؟
- هل أنشأت أي أدوات مخصصة من شأنها المساعدة في دعم احتياجات مشروعك المحدد، ولا سيما الأدوات التي تعمل على إجراء المهام المتكررة تلقائيًا؟
- بشكل عام، هل تستفيد بيئتك من الدعم المناسب للأدوات؟

نقاط مفتاحية

- يغفل المبرمجون أحيانًا بعضا من أقوى الأدوات لسنوات قبل اكتشافها.
- يمكن للأدوات الجيدة أن تجعل حياتك أسهل بكثير.
- تتوفر الأدوات بسهولة للتحرير، وتحليل جودة الشفرة، وإعادة التصنيع، والتحكم في الإصدار، وتصحيح الأخطاء، والاختبار، وضبط الشفرة.
- يمكنك صناعة العديد من الأدوات ذات الأغراض الخاصة التي تحتاج إليها.
- يمكن للأدوات الجيدة أن تقلل الجوانب الأكثر تعقيداً في تطوير البرمجيات، ولكنها لا تستطيع إلغاء الحاجة إلى البرمجة، لأنها سوف تستمر في إعادة تشكيل ما نعنيه بـ البرمجة".

القسم السابع: مهنة البرمجيات

في هذا القسم:

الفصل الحادي والثلاثون: التنسيق والأسلوب

الفصل الثاني والثلاثون: الشفرة الموثقة ذاتيا

الفصل الثالث والثلاثون: الميزة الشخصية

الفصل الرابع والثلاثون: موضوعات في مهنة البرمجيات

الفصل الخامس والثلاثون: أين تجد معلومات إضافية

الفصل 31

التنسيق والأسلوب

المحتويات[1]

مواضيع ذات صلة

- الشفرة الموثقة ذاتياً: الفصل 32
- أدوات تنسيق الشفرة: "التحرير" في القسم 30. 2

يستدير هذا الفصل إلى الجانب الجمالي من برمجة الحاسوب: تنسيق شفرة البرنامج المصدرية. إن المتعة المرئية والفكرية للشفرة المنسقة بشكل جيد هي لذة يستطيع قلة من المبرمجين تقديرها. ويستمد المبرمجون الذين يفتخرون بعملهم رضى جمالياً عظيماً من تزين البنيان المرئي لشفرتهم.

لا تؤثر التقنيات المذكورة في هذا الفصل على سرعة التنفيذ، أو استخدام الذاكرة، أو الجوانب الأخرى من البرنامج المرئية من خارج البرنامج. إنها تؤثر على سهولة فهم الشفرة، ومراجعتها، وتنقيحها بعد أشهر من كتابتها. إنها تؤثر أيضاً على سهولة قراءتها من قبل الآخرين، وفهمها، وتعديلها عندما تكون أنت خارج الصورة.

هذا الفصل مليء بالتفاصيل المنتقاة التي يشير إليها البشر عندما يتكلمون عن "الانتباه إلى التفاصيل." خلال حياة المشروع، التركيز على هكذا تفاصيل يصنع فرقاً في الجودة الأولية وقابلية الصيانة النهائية للشفرة التي

[1] cc2e.com/3187

تكتبها. تفاصيل كهذه متكاملة جداً مع عملية كتابة الشفرة، فمن الصعب أن تُغيَّر بشكل فعال لاحقاً. إن كانت ستُنجَز بكل الأحوال، فيتحتم أن تُنجَز خلال البناء الأولي. إن كنت تعمل على مشروع جماعي، دع فريقك يقرأ هذا الفصل ويتفق على أسلوب جماعي قبل أن تبدؤوا بكتابة الشفرة.

قد لا توافق على كل شيء تقرؤه هنا، لكن غايتي بأن أكسب موافقتك هي أدنى من أن أقنعك بأن تفكر بالقضايا المحتواة في أسلوب التنسيق. إن كان ضغط دمك عال، انتقل إلى الفصل التالي-إنه أقل جدلية.

31. 1 أساسيات التنسيق

يشرح هذا القسم نظرية التنسيق الجيد. وتشرح بقية هذا الفصل التطبيق.

تنسيقات متطرفة

اعتبر الإجرائية الظاهرة في التعداد 31-1:

التعداد 31-1 مثال على تنسيق جافا رقم 1

شفرة مرعبة

```
/* Use the insertion sort technique to sort the "data" array in
ascending order. This routine assumes that data[ firstElement ] is
not the first element in data and that data[ firstElement-1 ] can
be accessed. */ public void InsertionSort( int[] data, int
firstElement, int lastElement ) { /* Replace element at lower boundary
with an element guaranteed to be first in a sorted list. */ int
lowerBoundary = data[ firstElement-1 ]; data[ firstElement-1 ] =
SORT_MIN; /* The elements in positions firstElement through
sortBoundary-1 are always sorted. In each pass through the loop,
sortBoundary is increased, and the element at the position of the new
sortBoundary probably isn't in its sorted place in the array, so it's
inserted into the proper place somewhere between firstElement and
sortBoundary. */ for ( int sortBoundary = firstElement+1; sortBoundary
<= lastElement; sortBoundary++ ) { int insertVal = data[ sortBoundary
]; int insertPos = sortBoundary; while ( insertVal < data[ insertPos-1
] ) { data[ insertPos ] = data[ insertPos-1 ]; insertPos = insertPos-1;
} data[ insertPos ] = insertVal; } /* Replace original lower-boundary
element */ data[ firstElement-1 ] = lowerBoundary; }
```

الإجرائية صحيحة قواعدياً. إنها موثقة بتعليقات بعمق وتمتلك أسماء متحولات جيدة ومنطقاً واضحاً. إن كنت لا تصدق ذلك، اقرأها وجد خطأ! ما لا تمتلكه هذه الإجرائية هو التنسيق الجيد. هذا مثال متطرف، يسعى نحو "اللانهاية السالبة" في عدد الأسطر السيئة التنسيق إلى الجيدة التنسيق. التعداد 31-2 مثال أقل تطرفاً:



التعداد 31-2 مثال تنسيق جافا رقم 2

```
/* Use the insertion sort technique to sort the "data" array in
ascending order. This routine assumes that data[ firstElement ] is not
the first element in data and that data[ firstElement-1 ] can be
accessed. */
public void InsertionSort(
int[] data,
int firstElement,
int lastElement
) {
/* Replace element at lower boundary with an element guaranteed to be
first in a sorted list. */
int lowerBoundary = data[ firstElement-1 ];
data[ firstElement-1 ] = SORT_MIN;
/* The elements in positions firstElement through sortBoundary-1 are
always sorted. In each pass through the loop, sortBoundary is
increased, and the element at the position of the new sortBoundary
probably isn't in its sorted place in the array, so it's inserted into
the proper place somewhere between firstElement and sortBoundary. */
for (
int sortBoundary = firstElement+1;
sortBoundary <= lastElement;
sortBoundary++
) {
int insertVal = data[ sortBoundary ];
int insertPos = sortBoundary;
while ( insertVal < data[ insertPos-1 ] ) {
data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1;
}
data[ insertPos ] = insertVal;
}
/* Replace original lower-boundary element */
data[ firstElement-1 ] = lowerBoundary;
}
```

هذه الشفرة هي نفسها شفرة التعداد 31-1. مع أن معظم البشر سيوافقون على أن تنسيق الشفرة أفضل بكثير منه في المثال الأول، لا تزال الشفرة صعبة القراءة قليلاً. لا يزال التنسيق مزدحم ولا يقدم أي فكرة عن تنظيم الإجرائية المنطقي. إنه بحوالي 0 في عدد الأسطر السيئة التنسيق إلى الجيدة التنسيق. المثال الأول كان

مُختلقاً، لكن الثاني ليس غريباً على الإطلاق. رأيت برامجاً بطول عدة آلاف من الأسطر بتنسيق على الأقل بسوء هذا. بدون أي توثيق وبأسماء متحولات سيئة، كانت سهولة القراءة الكلية أدنى منها في هذا المثال. هذه الشفرة منسَّقة للحاسوب؛ لا توجد أي بيّنة أن الكاتب توقّع أن تُقرأ الشفرة من قبل البشر. التعداد 31-3 هو تحسين لها.



التعداد 31-3 مثال على تنسيق جافا رقم 3

```
/* Use the insertion sort technique to sort the "data" array in
ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed.
*/
public void InsertionSort( int[] data, int firstElement, int
lastElement )
// Replace element at lower boundary with an element guaranteed to be
// first in a sorted list.
int lowerBoundary = data[ firstElement-1 ];
data[ firstElement-1 ] = SORT_MIN;
/* The elements in positions firstElement through sortBoundary-1 are
always sorted. In each pass through the loop, sortBoundary
is increased, and the element at the position of the
new sortBoundary probably isn't in its sorted place in the
array, so it's inserted into the proper place somewhere
between firstElement and sortBoundary.
*/
for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
sortBoundary++ ) {
int insertVal = data[ sortBoundary ];
int insertPos = sortBoundary;
while ( insertVal < data[ insertPos - 1 ] ) {
data[ insertPos ] = data[ insertPos - 1 ];
insertPos = insertPos - 1;
}
data[ insertPos ] = insertVal;
}
// Replace original lower-boundary element
data[ firstElement - 1 ] = lowerBoundary;
}
```

تنسيق الإجرائية هذا موجب جداً في عدد الأسطر السيئة التنسيق إلى الجيدة التنسيق. الإجرائية الآن مبسوطة حسب المبادئ المشروحة في هذا الكتاب. أصبحت الإجرائية أكثر قابلية للقراءة، والجهد الذي بُذل لبناء التنسيق الجيد والتسمية الجيدة للمتحولات هو الآن واضح. كانت أسماء المتحولات تماماً بجودة ما كانت في الأمثلة السابقة، لكن التنسيق كان رديئاً جداً لدرجة أنها لم تكن مفيدة.

الفرق الوحيد بين هذا المثال والاثنين الأولين هو استخدام المسافات الفارغة- تُعامل الشفرة والتعليقات بالضبط نفس الشيء. المسافات الفارغة هي للاستخدام من أجل القراء البشر فقط-يستطيع حاسوبك أن يفسر أي من القطع الثلاثة السابقة بنفس السهولة. لا تشعر بالسوء من كونك لا تستطيع أن تقوم بأشياء بجودة حاسوبك!

النظرية الأساسية في التنسيق



النظرية الأساسية في التنسيق تقول إن التنسيق المرئي الجيد يُظهر البنيان المنطقي للبرنامج. أن تجعل الشفرة تبدو جميلة يستحق شيئاً ما، لكنه يستحق أقل من إظهار بنيان الشفرة. إن كانت إحدى التقنيات تُظهر البنيان إظهاراً أفضل من أخرى مظهرها أفضل، استخدم التي تُظهر البنيان إظهاراً أفضل. هذا الفصل يقدم أمثلة عديدة عن أساليب التنسيق التي تبدو جيدة لكنها تسيئ تمثيل التنظيم المنطقي للشفرة. في التطبيق، إن إعطاء الأولوية للتمثيل المنطقي لا يخلق شفرة بشعة عادة-مالم يكن منطق الشفرة بشعاً. التقنيات التي تجعل الشفرة الجيدة تبدو جيدة والشفرة السيئة تبدو سيئة هي أكثر فائدة من التقنيات التي تجعل كل الشفرات تبدو جيدة.

تفسيرَي الانسان والحاسوب لبرنامج[1]

التنسيق هو تلميح مفيد على بنيان البرنامج. بينما يهتم الحاسوب بشكل حصري بالأقواس أو begin وend، القارئ البشري ميال لرسم أفكار مستوحاة من التقديم المرئي للشفرة. فكر بمقطع الشفرة في التعداد 31-4، والتي فيها مخطط المسافات البادئة يجعلها تبدو للإنسان وكأن العبارات الثلاثة تُنفَّذ في كل مرة تُنفَّذ فيها الحلقة.

التعداد 31-4 **مثال جافا عن تنسيق يخبر قصة للبشر وأخرى مختلفة للحواسيب**

```
// تبديل عناصر اليمين و اليسار لكامل المصفوفة
for ( i = 0; i < MAX_ELEMENTS; i++ )
   leftElement = left[ i ];
   left[ i ] = right[ i ];
   right[ i ] = leftElement;
```

إن لم تحتوي الشفرة أقواس مطوَّقة، سينفذ المترجم العبارة الأولى *MAX_ELEMENTS* مرة والعبارتين الثانية والثالثة مرة واحدة. تجعل المسافة البادئة فكرة أن كاتب الشفرة أراد أن تُنفَّذ هذه العبارات الثلاثة كلها معاً وفكرة أنه نوى أن يضع أقواساً حولها أمراً واضحاً لي ولك. هذا لن يكون واضحاً للمترجم. التعداد 31-5 هو مثال آخر:

[1] يستطيع أي أحمق أن يكتب شفرة يستطيع الحاسوب فهمها. المبرمجون الجيدون هم الذين يكتبون شفرة يستطيع البشر فهمها. --مارتن فولير

التعداد 5-31 **مثال جافا آخر عن تنسيق يخبر قصة للبشر وأخرى مختلفة للحواسيب.**

```
x = 3+4 * 2+7;
```

يميل القارئ البشري لهذه الشفرة إلى تفسير العبارة لتعني أن القيمة (3+4)*(2+7)، أو 63 تُسند إلى x. سيتجاهل الحاسوب المسافات الفارغة ويطيع قواعد الأسبقية، مفسراً التعبير ب 3 + (4*2) + 7، أو 18. النقطة هي أن التنسيق الجيد يجعل البنيان المرئي لبرنامج يطابق البنيان المنطقي، أو يخبر البشر نفس القصة التي يخبرها للحواسيب.

كم يستحق التنسيق الجيد؟

تدعم دراساتنا ادعاء أن المعرفة بمخططات البرمجة وقواعد محاضرات البرمجة تستطيع أن تمتلك تأثيراً ملحوظاً على فهم البرنامج. في كتابهما المدعو [ال] العناصر الخاصة بأسلوب [البرمجة]، عرّف كيرنيغان وبلاوغير ما سندعوه قواعد المحاضرة. نتائجنا التجريبية تؤكد هذه القواعد: الأمر ليس مجرد قضية الجماليات حيث ينبغي أن تُكتب البرامج بأسلوب محدد. بالأحرى هناك قاعدة نفسية لكتابة برامج بحالة مألوفة: يمتلك المبرمجون توقعات قوية أن المبرمجين الآخرين سيتبعون قواعد المحاضرة هذه. إن انتهكت هذه القواعد، فإن الفائدة الاستخدامية المعطاة من قبل التوقعات التي بناها المبرمجون عبر الزمن تصبح باطلة فعلاً. نتائج التجارب على المبرمجين الطلاب المتقدمين والأغرار وعلى المبرمجين المحترفين الموصوفة في هذه المقالة تؤمن دعماً واضحاً لهذه الادعاءات.

-إيليوت سولو وي وكيت اهرليتش

في التنسيق[1]، ربما أكثر من أي جانب آخر للبرمجة، يظهر الفرق بين التواصل مع الحاسوب والتواصل مع البشر في اللعبة. الجزء الصغير من مهنة البرمجة هو كتابة برنامج بحيث يستطيع الحاسوب قراءته؛ والجزء الكبير هو كتابة برنامج بحيث يستطيع البشر الآخرون قراءته.

في مقالتهما التقليدية "الحدس في الشطرنج،" كتب شيس وسيمون تقريراً عن دراسة قارنت قدرات الخبراء والأغرار على تذكر مواقع قطع في الشطرنج (1973). عندما كانت القطع مرتبة على اللوح كما يمكن أن تكون خلال لعبة، ذاكرات الخبراء كانت متفوقة إلى حد بعيد على ذاكرات الأغرار. عندما رُتّبت القطع بشكل عشوائي، كان هناك فرق طفيف بين ذاكرات الخبراء والأغرار. التفسير التقليدي لهذه النتيجة هو أن ذاكرة الخبير ليست

[1] **إشارة مرجعية** التنسيق الجيد هو أحد مفاتيح قابلية القراءة. لتفاصيل حول قيمة قابلية القراءة، الق نظرة على القسم 34 .3، "اكتب برامجاً للبشر أولاً، وللحواسيب ثانياً.

بشكل وراثي أفضل من ذاكرة الغرّ، بل إن الخبير لديه بنيان معرفي يساعده بتذكر أنواع محددة من المعلومات. عندما تتوافق المعلومة الجديدة مع هذا البنيان المعرفي-في هذه الحالة، التموضع ذو المعنى لقطع الشطرنج- يستطيع الخبير تذكرها بسهولة. عندما لا تتوافق المعلومة الجديدة مع البنيان المعرفي-قطع الشطرنج موزعة بشكل عشوائي-لا يستطيع الخبير تذكرها بشكل أفضل من الغرّ.

بعد عدة سنوات، ضاعف شنيدرمان نتائج شيس وسيمون في حلبة برمجة الحاسوب وكتب نتائجه في مقالة سُمّيت "تجارب استكشافية في سلوك المبرمج" (1976). وجد شنيدرمان أنه عندما كانت عبارات البرنامج مرتبة بترتيب ذي معنى، كان الخبراء قادرون على تذكرها بشكل أفضل من الأغرار. عندما كانت العبارات مبعثرة، تناقصت أفضلية الخبراء. أكّدت نتائج شنيدرمان بدراسات أخرى (ماك كيثين وآخرون 1981، سولو وي واهرليتش 1984). أكّد المفهوم الأساسي أيضاً في ألعاب **غو** والجسر وفي الالكترونيات، والموسيقى، والفيزياء (ماك كيثين وآخرون 1981).

بعد أن أصدرت النسخة الأولى من هذا الكتاب، هانك، أحد المبرمجين الذين راجعوا النسخة المكتوبة باليد، قال "كنت متفاجئاً من أنك لم تحاجج بقوة أكثر في تفضيل أسلوب الأقواس الذي يبدوا كهذا:

```
for ( ... )
   {
   }
```

"كنت متفاجئاً من أنك ضمَّنت أسلوب الأقواس الذي يبدو كالتالي:

```
for ( ... )   {
}
```

"كنت أعتقد أنه، مع دفاعي أنا وطوني عن الأسلوب الأول، ستفضله."

أجبت، "أنت تعني أنك كنت تدافع عن الأسلوب الأول، وطوني كان يدافع عن الأسلوب الثاني، أليس كذلك؟ دافع طوني عن الأسلوب الثاني، وليس الأول."

أجاب هانك، "هذا مضحك. آخر مشروع عملت فيه أنا وطوني سوية، فضّلت الأسلوب رقم 2، وطوني فضّل الأسلوب رقم 1. لقد قضينا كل المشروع ونحن نتجادل حول أي أسلوب هو الأفضل. أعتقد أننا ناقشنا بعضنا حتى فضّل كلانا أسلوب الآخر!"



هذه التجربة، بالإضافة إلى الدراسات المُقتبَسة فوق، تقترح أن الهيكلة تساعد الخبراء في حدس، وفهم، وتذكّر الميزات المهمة في البرامج. غالباً ما يلتصق المبرمجون الخبراء بأساليبهم الخاصة بعناد، حتى عندما تكون مختلقة بشكل شاسع عن أساليب أخرى مُستخدمة من قبل مبرمجين خبراء آخرين. الخلاصة هي أن تفاصيل طريقة محددة في هيكلة البرنامج هي أقل أهمية بكثير من حقيقة أن البرنامج مُهيكَل بشكل متماسك.

التنسيق والدين

أهمية فهم وتذكّر الهيكلة بطريقة مألوفة لبيئة الشخص قادت بعض الباحثين إلى أن يفترضوا أن التنسيق قد يؤذي قدرة الخبير على قراءة برنامج إن كان التنسيق مختلفاً عن المخطط الذي يستخدمه الخبير (شيل 1981، سولو وي واهرليتش 1984). هذه الاحتمالية، تزداد بحقيقة أن التنسيق هو تمرين جمالي بالإضافة إلى أنه منطقي، هذا يعني أن الجدالات حول تنسيق البرنامج غالباً ما تبدو مشابهة للحروب الدينية أكثر من النقاشات الفلسفية.



على صعيد خشن، من الواضح أن بعض صيغ التنسيق أفضل من بعض[1]. التنسيقات المتفاضلة بالتتابع لنفس الشفرة في بداية هذا الفصل يجعل هذا الأمر بيّناً. لن يتجنب هذا الكتاب النقاط الأجود من غيرها في التنسيق فقط لأنها جدلية. ينبغي أن يكون المبرمجون الجيدون ذوي فكر مفتوح حول تطبيقات تنسيقهم وأن يقبلوا التطبيقات المُثبت أنها أفضل من التطبيقات التي اعتادوا عليها، حتى وإن كان ينتج عن التكيّف على منهجية جديدة بعض الإزعاج في البداية.

أهداف التنسيق الجيد[2]

العديد من القرارات المتعلقة بتفاصيل التنسيق مرتبط بقضايا شخصية في الجماليات؛ غالباً، تستطيع أن تنجز نفس الهدف بعدة طرق، تستطيع أن تجعل الجدالات عن القضايا الشخصية أقل التصاقاً بالذات إن حددت بشكل واضح معياراً للأفضلية الخاصة بك. ومن ثم، وكما هو واضح، ينبغي أن يقوم تخطيط تنسيق جيد بالتالي:

[1] **إشارة مرجعية** إن كنت تجمع بين البرمجيات والدين، قد تقرأ القسم 34. 9، "افصل بقوة بين البرمجيات والدين" قبل أن تقرأ بقية هذا الفصل.

[2] تشير النتائج إلى قابلية كسر الخبرة البرمجية: يمتلك المبرمجون المتقدمون توقعات قوية عما ينبغي للبرنامج أن يبدو عليه، وعندما تُنتهك هذه التوقعات-بطريقة يبدو أنها غير مؤذية-فإن أدائهم ينخفض بشكل فظيع. -إيليوت سولو وي وكيت اهرليتش

تمثيل البنيان المنطقي للشفرة بدقة هذه هي النظرية الأساسية في التنسيق مجدداً: الغرض الأساسي من التنسيق الجيد هو أن يُظهر البنيان المنطقي للشفرة. عادةً، يستخدم المبرمجون المسافات البادئة وأنواع المسافات الفارغة الأخرى ليظهروا البنيان المنطقي.

تمثيل البنيان المنطقي للشفرة بشكل متماسك تمتلك بعض أساليب التنسيق قواعداً فيها العديد من الاستثناءات التي من الصعب أن تتبع القواعد بشكل متماسك. الأسلوب الجيد يُطبَّق على معظم الحالات.

يحسّن قابلية القراءة استراتيجية مسافات بادئة منطقية ولكنها تجعل الشفرة أصعب قراءةً هي استراتيجية عديمة الفائدة. مخطط تنسيق يستدعي الفراغات فقط عندما يطلبها المترجم هو مخطط منطقي لكنه غير مقروء. يجعل مخطط التنسيق الجيد الشفرة أسهل قراءةً.

الصمود مقابل التعديلات تبقى مخططات التنسيق الأفضل صامدة تحت تعديلات الشفرة. لا ينبغي أن يتطلب تعديل سطر واحد من الشفرة تعديل عدة أسطر أخرى.

بالإضافة إلى هذا المعيار، تصغير عدد أسطر الشفرة اللازمة لتحقيق عبارة بسيطة أو كتلة هو أمر مُعتَبَر في بعض الأحيان.

كيف تضع أهدافاً تنسيقية للاستخدام



بإمكانك استخدام معيار لمخطط التنسيق الجيد لتأرّض نقاشاً حول التنسيق بحيث تظهر الأسباب الذاتية لتفضيل أسلوب على آخر للعلن.

وزن المعيار بطرق مختلفة قد يؤدي إلى خلاصات مختلفة. مثلاً، إن كنت تشعر بقوة أن تصغير عدد الأسطر المستخدمة على الشاشة هو مهم-ربما لأنك تملك شاشة حاسوب صغيرة-قد تنتقد أسلوباً لأنه يستخدم سطرين زيادة على أسلوب آخر من أجل لائحة وسطاء إجرائية.

31. 2 تقنيات التنسيق

تستطيع أن تحقق تنسيقاً جيداً باستخدام أدوات تنسيق قليلة بعدة طرق مختلفة. هذا القسم يصف كل واحدة منها.

المسافات الفارغة

استخدمالمسافاتالفارغةلتحسّنقابليةالقراءة. المسافات الفارغة، متضمنةً الفراغات، وعلامات الجدولة، وعلامات انتهاء السطر، والأسطر الفارغة، هي الأداة الرئيسة المتاحة لك لتُظهر بنيان البرنامج.

إشارة مرجعية اكتشف بعض الباحثون التشابه بين هيكلة كتاب وهيكلة برنامج. لمعلومات، انظر "نموذج الكتاب لتنسيق البرنامج" في القسم 32. 5.لن تفكر بكتابة كتاب بدون فراغات بين الكلمات، وبدون علامات انتهاء للفقرات، وبدون تقسيمه إلى فصول. هكذا كتاب يمكن أن يُقرأ من الغلاف إلى الغلاف، لكن سيكون فعلياً من المستحيل أن تستخلص منه مسار تفكير أو أن تجد فيه فقرة مهمة. ربما أكثر أهمية، لن يُظهر تنسيق الكتاب للقارئ كيف نوى الكاتب أن ينظم المعلومات. تنظيم الكاتب هو دليل مهم على التنظيم المنطقي للمواضيع.

تجزيء الكتاب إلى فصول وفقرات وجمل يظهر للقارئ انتظام الموضوع فكرياً. إن كان التنظيم غير واضح، فعلى القارئ أن يؤمن التنظيم، والذي يضع حملاً أكبر بكثير على القارئ ويضيف إمكانية عجز القارئ عن اكتشاف كيفية تنظيم الموضوع..

المعلومات الموجودة في برنامج أكثف من المعلومات الموجودة في معظم الكتب. بينما قد تقرأ وتفهم صفحة من كتاب في دقيقة أو اثنتين، لا يستطيع معظم المبرمجين أن يقرؤوا ويفهموا تعداداً من صفحة من برنامج مجرّد في أي معدل قريب إلى ذاك. ينبغي أن يعطي البرنامج أفكاراً عن التنظيم أكثر مما يفعل الكتاب وليس أقل.

التجميع من الناحية الأخرى للمرآة، المسافات الفارغة هي مُجمِّعات، تؤكد أن العبارات المترابطة مجتمعة مع بعضها.

في الكتابة، الأفكار تُجمّع في فقرات. تحتوي الفقرة المكتوبة بشكل جيد جملاً متصلة بفكرة محددة. لا ينبغي أن تحتوي جملاً خارجة عن الموضوع. بشكل مشابه، ينبغي أن تحتوي فقرة من الشفرة عبارات تنجز مهمة بسيطة وهذه العبارات مرتبطة ببعضها البعض.

السطور الفارغة تماماً بمقدار أهمية تجميع العبارات المترابطة، هي أهمية فصل العبارات غير المترابطة عن بعضها البعض. بداية فقرة جديدة في **الإنكليزية** تُعرف بمسافة بادئة أو سطر فارغ. ينبغي أن تُعرف بداية فقرة جديدة من الشفرة بسطر فارغ.

استخدام السطور الفارغة هو طريقة لتحديد كيفية تنظيم البرنامج. تستطيع أن تستخدمها لتوزيع مجموعات من العبارات المترابطة إلى فقرات، ولتفصل الإجرائيات عن بعضها البعض، وتسلط الضوء على التعليقات.

بيانات مثبتة

مع أن هذه الإحصائية قد تكون صعبة الوضع قيد التنفيذ، وجدت دراسة من قبل غورلا وبيناندر، وبيناندر أن العدد المثالي للأسطر الفارغة في برنامج ينبغي أن يكون بين 8 و16 بالمئة. وفوق 16 بالمئة، يزداد زمن التصحيح بشكل فظيع (1990).

المسافات البادئة استخدم المسافات البادئة لتُظهر البنيان المنطقي للبرنامج. كقاعدة، ينبغي أن تضع مسافات بادئة للعبارات تحت العبارة التي ينتمينَ إليها.

بيانات مثبتة

قد ظهر أن المسافات البادئة مترابطة مع فهم المبرمج المتزايد. كتبت المقالة "المسافات البادئة للبرنامج وقابلية الفهم" أن عدة دراسات وجدت ترابطاً بين المسافات البادئة والفهم المتحسَّن (مياريا وآخرون 1983). حققت مواضيع في اختبار للفهم، عندما كانت البرامج تحتوي مخطط مسافات بادئة من فراغين إلى أربعة، من 20 إلى 30 بالمئة أكثر مما حققته عندما كانت البرامج لا تحتوي مسافات بادئة على الإطلاق.

بيانات مثبتة

وجدت نفس الدراسة أنه من المهم ألا تقلل التأكيد ولا تزيد التأكيد على بنيان البرنامج المنطقي. نتائج الفهم الأقل كانت في البرامج التي لم تحتوي مسافات بادئة على الإطلاق. النتائج الثانية في القلّة كانت للبرامج التي استخدمت مسافات بادئة بستة فراغات. خلصت الدراسة إلى أن مسافات بادئة من فراغين إلى أربعة كانت مثالية. وبشكل ممتع، الكثير من المواضيع في التجربة أعطت شعوراً أن المسافات البادئة بستة فراغات كانت أسهل الاستخدام من المسافات الفارغة الأصغر، حتى وإن كانت نتائجها أدنى. هذا ربما لأن المسافات البادئة بستة فراغات تبدو مُحبّبة. لكن بغض النظر عن كم تبدو جميلة، تنقلب المسافات البادئة بستة فراغات لتكون أقل قابلية للقراءة. هذا مثال عن التصادم بين المطلب الجمالي وقابلية القراءة.

الأقواس

استخدم أقواس أكثر مما تعتقد أنك تحتاج. استخدم أقواساً لتوضح التعابير التي تحتوي أكثر من حدّين. قد لا تكون لازمة، لكنها تضيف وضوحاً ولا تكلّفك أي شيء. مثلاً، كيف تُقيَّم التعابير التالية؟

إصدار سي ++: 8 / 7 *3 % 12 + 4

إصدار مايكروسوفت فيجوال بيسك: `mod 3 * 7 / 8 12 + 4`

السؤال المفتاحي هو، هل عليك أن تفكر كيف سيُقيَّم التعبير؟ هل أنت واثق بإجابتك ولا تحتاج أن تفحص بعض المراجع؟ حتى المبرمجون الخبراء لا يجاوبون بثقة، وهذا لمَ ينبغي عليك أن تستخدم أقواساً متى وجد شك في كيفية تقييم التعابير.

31. 3 أساليب التنسيق

يجب على معظم قضايا التنسيق أن تتعامل مع تنسيق الكتل، مجموعات العبارات تحت عبارات التحكم. الكتلة محاصرة بين قوسين أو كلمتين مفتاحيتين: { و } في سي ++ وجافا و if-then-endif في فيجوال بيسك، وتراكيب أخرى مشابهة في لغات أخرى. للتبسيط، تستخدم معظم هذه النقاشات begin وend بشكل شامل، مفترضةً أنك تستطيع أن تستنتج كيفية تطبيق النقاش على الأقواس في سي ++ وجافا وآليات التكتيل الأخرى في اللغات الأخرى. تصف الأقسام التالية أربعة أساليب عامة للتنسيق:

- الكتل النقية
- محاكاة الكتل النقية
- استخدام أزواج begin-end (القوسين المعقوفين) لتعيين حدود الكتلة
- تنسيق خط النهاية

الكتل النقية

تُساق العديد من الجدالات حول التنسيق من الارتباك الموروث في لغات البرمجة الأكثر شيوعاً. تمتلك اللغة المُصمَّمة بشكل جيد هياكل كتل واضحة والتي تضفي على نفسها أسلوب المسافات البادئة الطبيعي. في فيجوال بيسك، مثلاً، يمتلك كل تركيب تحكم مُنهيه الخاص ولا تستطيع استخدام تركيب التحكم بدون استخدام المنهي. الشفرة مقسمة إلى كتل بشكل طبيعي. تظهر بعض الأمثلة في فيجوال بيسك في التعدادات 31-6 و31-7 و31-8:

التعداد 31-6 مثال فيجوال بيسك عن كتلة if نقية.

```
If pixelColor = Color_Red Then
   statement1
   statement2
   ...
End If
```

التعداد 31-7 مثال فيجوال بيسك عن كتلة while نقية.

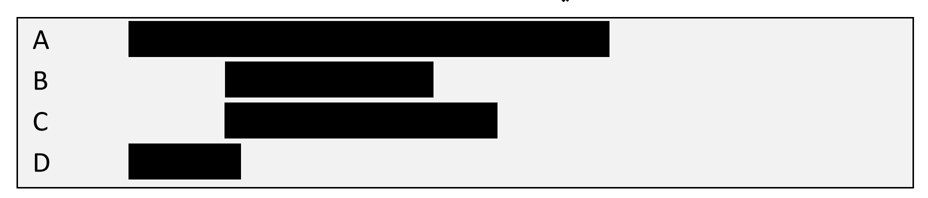
```
while pixelColor = Color_Red
   statement1
   statement2
   ...
wend
```

التعداد 31-8 مثال فيجوال بيسك عن كتلة case نقية.

```
Select Case pixelColor
   Case Color_Red
      statement1
      statement2
      ...
   Case Color_Green
      statement1
      statement2
      ...
   Case Else
      statement1
      statement2
      ...
End Select
```

لتركيبة التحكم في **فيجوال بيسك** عبارة بداية دوماً-*if-then* و *while* و*select-case* في الأمثلة-ولها عبارة *نهاية* موافقة دوماً. وضع مسافات بادئة للجزء الداخلي من التركيبة ليس تطبيقاً خلافياً، والخيارات المتاحة لمحاذاة الكلمات المفتاحية الأخرى نوعاً ما محدودة. التعداد 31-9 هو تمثيل تجريدي لكيفية عمل هذا النوع من التنسيق:

التعداد 31-9 **مثال تجريدي عن أسلوب التنسيق الكتلة النقية**



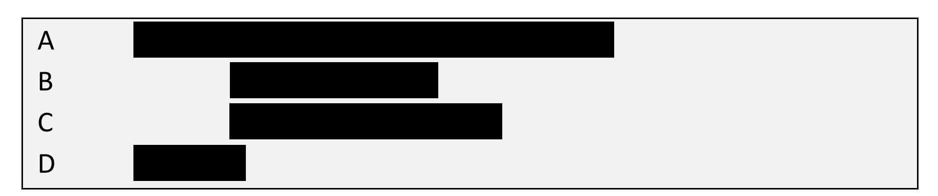
في هذا المثال، العبارة A تُبدئ تركيبة التحكم والعبارة D تنهيها. تؤمن المحاذاة بين العبارتين إغلاقاً مرئياً متماسكاً.

ظهر الجدل حول تنسيق بنى التحكم من حقيقة أن بعض اللغات لا *تتطلب* بنى كتلية. يمكن أن يكون لديك *if-then* متبوعة بعبارة مفردة ولا تمتلك كتلة رسمية. عليك أن تضيف زوج *begin-end* أو قوسي بداية ونهاية معقوفين لتخلق الكتلة بدلاً من الحصول على واحدة تلقائياً مع كل تركيبة تحكم. فك اقتران *begin* و *end* عن تركيبة التحكم-كما تفعل لغات مثل **سي** ++ و **جافا** مع { و}-يؤدي إلى أسئلة عن مكان وضع *begin* و *end*. إذن، العديد من مشاكل المسافات البادئة هي مشاكل فقط لأنه عليك أن تعوض عن بنى اللغة المصممة برداءة. طرق متنوعة للتعويض مشروحة في الأقسام التالية.

محاكاة الكتل النقية

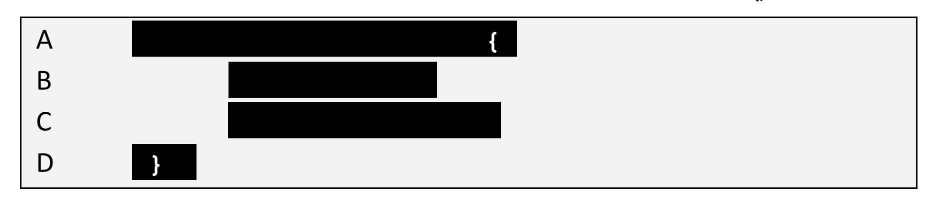
إظهار الكلمتين المفتاحيتين begin و end (أو الرمزين { و }) كامتداد لتركيبة التحكم التي تستخدمها هو نهج جيد في اللغات التي لا تمتلك كتلاً نقية. إنه أمر ذو معنى أن تحاول أن تحاكي تنسيق فيجوال بيسك في لغتك. التعداد 31-10 هو منظر تجريدي للبنيان المرئي الذي تحاول أن تحاكيه:

التعداد 31-10 **مثال تجريدي عن أسلوب التنسيق الكتلة النقية**



في هذا الأسلوب، تفتح تركيبة التحكم الكتلة في العبارة A وتنهي الكتلة في العبارة D. هذا يقتضي أن *begin* ينبغي أن تكون موجودة في نهاية العبارة A وend ينبغي أن تكون العبارة D. بالتجريد، لتحاكي الكتل النقية، عليك أن تقوم بشيء ما مثل التعداد 31-11:

التعداد 31-11 مثال تجريدي عن محاكاة أسلوب الكتلة النقية



بعض الأمثلة عما يبدو عليه الأسلوب في **سي**++، تظهر في التعدادات 31-12 و31-13 و31-14:

التعداد 31-12 مثال سي++ عن محاكاة كتلة if نقية.

```
if ( pixelColor == Color_Red ) {
   statement1;
   statement2;
   ...
}
```

التعداد 31-13 مثال سي ++ عن محاكاة كتلة while نقية.

```
while ( pixelColor == Color_Red ) {
   statement1;
   statement2;
   ...
}
```

التعداد 31-14 مثال سي++ عن كتلة switch/case نقية.

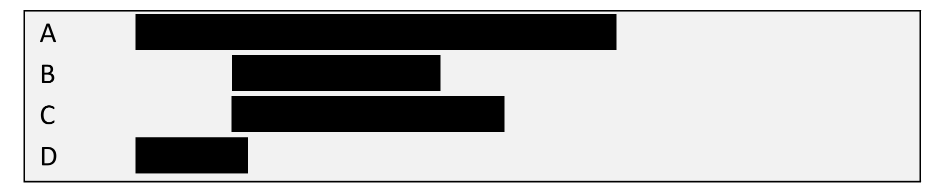
```
switch ( pixelColor ) {
   case Color_Red:
      statement1;
      statement2;
      ...
   break;
   case Color_Green:
      statement1;
      statement2;
      ...
   break;
   default:
      statement1;
      statement2;
      ...
   break;
}
```

أسلوب المحاذاة هذا يعمل تماماً بشكل جيد. مظهره جيد، تستطيع أن تطبّقه بشكل متماسك، وهو قابل للصيانة. إنه يدعم **النظرية الأساسية في التنسيق** في أنه يساعد على إظهار البنيان المنطقي للشفرة. إنه خيار منطقي للأسلوب. هذا الأسلوب هو المعياري في **جافا** والشائع في **سي** ++.

استخدام الزوجين *begin-end* (القوسين المعقوفين) لتعيين حدود الكتلة

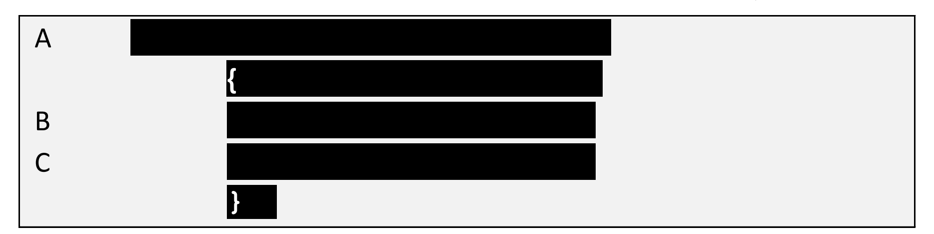
بديل لبنيان الكتلة النقية هو إظهار الزوجين begin-end كحدود للكتلة. (يستخدم النقاش التالي begin-end ليشير بشكل عام إلى الزوجين begin-end، والقوسين المعقوفين، وتراكيب اللغات المكافئة الأخرى.) إن اتخذت هذا النهج، فإنك تُظهر begin وend كعبارتين تتبعان التركيبة المنطقية بدلاً من كونهما جزأين من مكوناتها. بالنسبة للمظهر الرسومي، إنه مثالي، تماماً بمقدار ما كان في محاكاة الكتلة النقية الظاهرة مجدداً في التعداد 31-15:

التعداد 31-15 **مثال تجريدي لأسلوب التنسيق الكتلة النقية**



لكن في هذا الأسلوب، لتعامل *begin* و *end* كجزئين من بنيان الكتلة بدلاً من عبارة التحكم، عليك أن تضع *begin* في بداية الكتلة (وليس في نهاية عبارة التحكم) وتضع *end* في نهاية الكتلة (بدلاً من أن تُنهي عبارة التحكم). في هذا التجريد، عليك أن تقوم بشيء ما مثل المنجز في التعداد 31-16:

التعداد 31-16 **مثال تجريدي عن استخدام begin وend كحدود للكتلة.**



بعض الأمثلة عن كيفية استخدام *begin* و*end* كمنظر لحدود للكتلة في **سي++**، ظاهرة في التعداد 17-31 و18-31 و19-31:

التعداد 17-31 مثال سي ++ عن استخدام begin و end كحدود للكتلة في كتلة if.

```
if ( pixelColor == Color_Red )
   {
   statement1;
   statement2;
   ...
   }
```

التعداد 18-31 مثال سي ++ استخدام begin و end كحدود للكتلة في كتلة while.

```
while ( pixelColor == Color_Red )
   {
   statement1;
   statement2;
   ...
   }
```

التعداد 19-31 مثال سي ++ عن استخدام begin وend كحدود للكتلة في كتلة switch/case.

```
switch ( pixelColor )
   {
   case Color_Red:
      statement1;
      statement2;
      ...
      break;
   case Color_Green:
      statement1;
      statement2;
      ...
      break;
   default:
      statement1;
      statement2;
      ...
      break;
   }
```

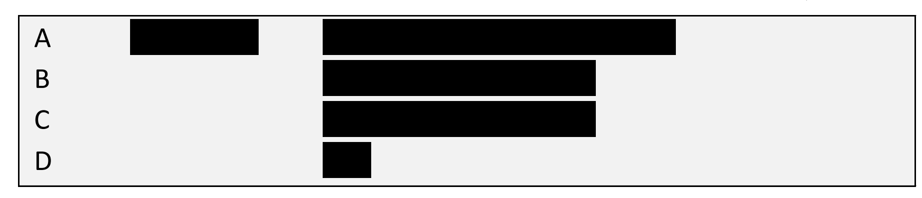
أسلوب المحاذاة هذا يعمل بشكل جيد؛ إنه يدعم **النظرية الأساسية في التنسيق** (مرة ثانية، بكشف البنيان المنطقي المُشكِّل للشفرة). محدوديته الوحيدة هي أنه لا يمكن أن يُطبَّق حرفياً في عبارات *switch/case* في **سي ++ وجافا**، كما يظهر في التعداد 19-31. (الكلمة المفتاحية *break* هي بديل لقوس الإغلاق، لكن لا يوجد مكافئ لقوس الفتح).

تنسيق خط النهاية

"تنسيق خط النهاية" هو استراتيجية تنسيق أخرى، والتي تشير إلى مجموعة كبيرة من استراتيجيات التنسيق فيها تُحاذى الشفرة إلى منتصف أو نهاية السطر. المسافات البادئة لخط النهاية تُستخدم لتُحاذي كتلة بكلمة

مفتاحية تبدؤها، لتجعل الوسطاء التابعة لإجرائية مُصطَّفة تحت الوسيط الأول، ولتصطِّف الحالات في عبارة case، ولأغراض أخرى مشابهة. التعداد 31-20 هو مثال تجريدي:

التعداد 31-20 مثال تجريدي عن أسلوب التنسيق خط النهاية



في هذا المثال، العبارة A تبدأ تركيبة التحكم والعبارة D تنهيها. العبارات B C D مُنتظَمة تحت الكلمة المفتاحية التي تبدأ الكتلة في العبارة A. تُظهر المسافات البادئة الموحدة لـ B C D أنها مجتمعة مع بعضها. التعداد 31-21 مثال أقل تجريد لشفرة منسقة باستخدام هذه الاستراتيجية:

التعداد 31-21 مثال فيجوال بيسك عن تنسيق خط النهاية لكتلة while.

```
while ( pixelColor = Color_Red )
          statement1
          statement2
          ...
          wend
```

في هذا المثال، *begin* موضوعة في نهاية السطر وليس تحت الكلمة المفتاحية الموافقة. يفضل بعض الناس وضع *begin* تحت الكلمة المفتاحية، لكن الاختيار بين هاتين النقطتين اللطيفتين هو أصغر مشاكل هذا الأسلوب.

يعمل أسلوب التنسيق خط النهاية بشكل مقبول في حالات قلّة. التعداد 31-22 هو مثال فيه يعمل بنجاح:

التعداد 31-22 مثال فيجوال بيسك نادر فيه يبدو أسلوب خط النهاية جذاباً

```
If ( soldCount > 1000 ) Then
                            markdown = 0.10
                            profit = 0.05
                        Else
                            markdown = 0.05
                        End If
```

الكلمة المفتاحية *else* مُنتظَمة مع الكلمة المفتاحية *then* فقا

في هذه الحالة، الكلمات المفتاحية"*then*، *else*، *if end*" منتظَمة والشفرة اللاحقة لها منتظَمة أيضاً. الأثر المرئي هو بنيان منطقي واضح.

إن نظرت بعين الناقد إلى مثال عبارة *case* السابق، من الممكن أن تتنبأ بفكّ هذا الأسلوب. عندما تصبح الشرطية أكثر تعقيداً، سيعطي هذا الأسلوب أفكاراً عديمة الفائدة أو مضلِّلة عن البنيان المنطقي. التعداد 31-23 هو مثال عن كيفية تحطم هذا الأسلوب عندما يُستخدم في شرطيات أعقد:

التعداد 31-23 مثال فيجوال بيسك أكثر نمطية، يتحطم فيه أسلوب خط النهاية.

```
If ( soldCount > 10 And prevMonthSales > 10 ) Then
   If ( soldCount > 100 And prevMonthSales > 10 ) Then
      If ( soldCount > 1000 ) Then
         markdown = 0.1
         profit = 0.05
      Else
         markdown = 0.05
      End If
                  Else
                     markdown = 0.025
                  Else
                     markdown = 0.0
               End If
```



ما هو سبب التنسيق الغريب لعبارتي else في نهاية المثال؟ إنهما مُنتظمتين بشكل متماسك تحت الكلمات المفتاحية الموافقة، لكن من الصعب أن تدافع عن أن انتظامهما بهذا الشكل يوضح البنيان المنطقي. وإذا عُدَّلت الشفرة بحيث يتغير طول السطر الأول، سيتطلب أسلوب خط النهاية تغيُّر المسافات البادئة للعبارات الموافقة. هذا يطرح مشكلة في الصيانة لا يطرحها أسلوب الكتلة النقية ولا أسلوب محاكاة الكتلة النقية ولا أسلوب استخدام begin-end لتعيين حدود الكتلة.

قد تعتقد أن هذه الأمثلة أُبدعت فقط لتسجيل نقطة، لكن هذا الأسلوب مستمر مع سيئاته. كتب ومراجع برمجية عديدة تنصح بهذا الأسلوب. أول كتاب رأيته ينصح بهذا الأسلوب نُشر في منتصف سبعينيات القرن الماضي، وأحدث كتاب نُشر في 2003.

بعد كل شيء، تنسيق خط النهاية غير دقيق، وصعب التطبيق بتماسك، وصعب الصيانة. سترى مشاكل أخرى في تنسيق خط النهاية عبر هذا الفصل.

أي أسلوب هو الأفضل

إن كنت تعمل على فيجوال بيسك، استخدم محاذاة الكتلة النقية. (بيئة فيجوال بيسك تجعل من الصعب ألا تستخدم هذا الأسلوب على أي حال.)

في جافا، التطبيق المعياري هو استخدام محاذاة الكتلة النقية.

في سي ++، بإمكانك ببساطة أن تختار الأسلوب الذي تحبه أو المفضل لأغلبية الناس في فريقك. إما محاكاة الكتلة النقية أو حدود الكتلة begin-end فهما يعملان بشكل جيد بتساوٍ. وجدت الدراسة الوحيدة التي قارنت الأسلوبين أنه لا فرق ملحوظ إحصائياً بين الاثنين طالما أن قابلية الفهم هي المَعنيّة (هانسن ويم 1987).

ولا واحد من الأساليب مقاوم للحمقى، وكل منها يتطلب تسوية "بين المنطق والمظهر" من حين لآخر. قد تفضل واحداً أو آخر لأسباب جمالية. يستخدم هذا الكتاب أسلوب الكتلة النقية في أمثلة الشفرة، لذا تستطيع أن ترى الكثير جداً من التوضيحات لكيفية عمل هذا الأسلوب فقط بالمرور عبر أمثلته. حالما تختار أسلوب، فإنك تحصد الفائدة العظمى من التنسيق الجيد عندما تطبقه بشكل متماسك.

31.4 رسم بنى التحكم

تنسيق بعض عناصر البرنامج هو بشكل رئيس قضية الجماليات.[1] يؤثر تنسيق بنى التحكم، على كل، على قابلية القراءة والفهم وهو لذلك أولوية تطبيقية.

نقاط حسنة لتنسيق كتل بنى التحكم

العمل مع كتل بنى التحكم يتطلب انتباهاً إلى بعض التفاصيل الدقيقة. هاهنا بعض التوجيهات:

تجنب أزواج* begin-end *غير المسبوقة بمسافات بادئة في الأسلوب الظاهر في التعداد 31-24، ينتظم الزوج *begin-end* مع تركيبة التحكم، والعبارات التي تغلق عليها *begin* و*end* سُبقت بمسافات بادئة تحت *begin*.

التعداد 31-24 مثال جافا عن زوجي begin-end غير مسبوقين بمسافات بادئة.

```
for ( int i = 0; I < MAX_LINES; i++ )
{
    ReadLine( i );
    ProcessLine( i );
}
```

begin منتظمة مع for

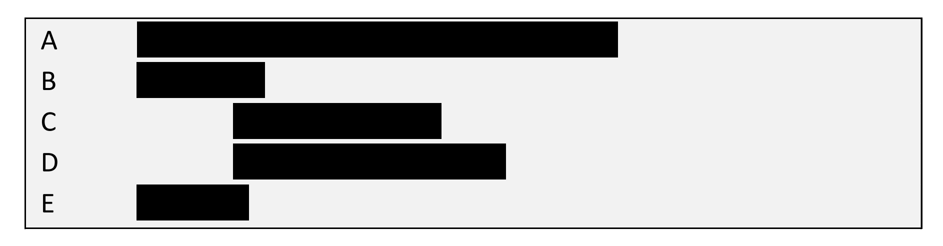
العبارات مسبوقة بمسافات بادئة تحت begin

end منتظمة مع for

مع أن هذا النهج يبدو جيداً، فإنه ينتهك **النظرية الأساسية في التنسيق**؛ إنه لا يُظهر البنيان المنطقي للشفرة. باستخدامه بهذه الطريقة، *begin* و *end* ليستا جزءاً من تركيبة التحكم، لكنهما ليستا جزءاً من العبارات التي بعدها أيضاً.

التعداد 31-25 هو منظر تجريدي لهذا النهج:

التعداد 31-25 مثال تجريدي لمسافات بادئة مضللة.



في هذا المثال، هل العبارة B تابعة للعبارة A؟ إنها لا تبدو كجزء من العبارة A، ولا تبدو أنها تابعة لها أيضاً. إن كنت تستخدم هذا النهج، بدّل إلى واحد من أسلوبي التنسيق المشروحين سابقاً وتنسيقك سيصبح أكثر تماسكاً.

[1] **إشارة مرجعية** لتفاصيل حول توثيق بنى التحكم، انظر "توثيق بنى التحكم" في القسم 32. 5. لنقاش حول الجوانب الأخرى لبنى التحكم، انظر الفصول من 14 إلى 19.

***تجنب المسافات البادئة المضاعفة لـ* begin و end** القاعدة المانعة لمضاعفة المسافات البادئة أمام زوجي *begin-end* هي نتيجة مباشرة للقاعدة المانعة لزوجي *begin-end* غير مسبوقين بمسافات بادئة. في هذا الأسلوب، الظاهر في التعداد 31-26، وضعت مسافات بادئة لـ*begin* و*end* ووضعت مسافات بادئة مجدداً للعبارات التي بينهما:

التعداد 31-26 **مثال جافا عن مسافات بادئة مضاعفة لكتلة begin-end.**



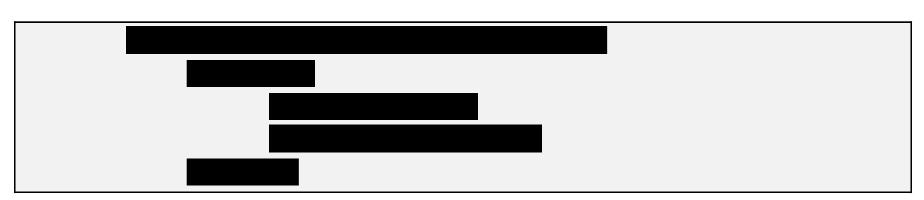
```
for ( int i = 0; i < MAX_LINES; i++ )
    {
        ReadLine( i );
        ProcessLine( i );
    }
```

العبارات تحت *begin*

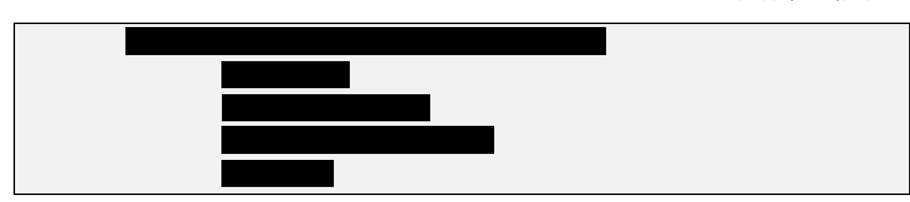
هذا مثال آخر على أسلوب يبدو جيداً لكنه يخرق **النظرية الأساسية في التنسيق**. أظهرت دراسة أن لا فرق بالفهم بين البرامج التي تستخدم مسافات بادئة مفردة والتي تستخدم مسافات بادئة مضاعفة (مياريا وآخرون. 1983)، لكن هذا الأسلوب لا يظهر بدقة البنيان المنطقي للبرنامج. *ReadLine()* و *ProcessLine()* يبدوان وكأنهما تابعتين منطقياً لزوج *begin-end*، لكنهما ليستا كذلك.

وكذلك يضخم هذا النهج تعقيد بنيان البرنامج المنطقي. أي التركيبتين الظاهرتين في التعدادين 31-27 و31-28 تبدو أكثر تعقيداً؟

التعداد 31-27 **تركيبة تجريدية 1.**



التعداد 31-28 **تركيبة تجريدية 2**



كلاهما تمثيل تجريدي لتركيبة حلقة *for*. التركيبة التجريدية 1 تبدو أكثر تعقيداً حتى وإن كانت تمثل نفس الشفرة التي تمثلها التركيبة التجريدية 2. إن كنت ستعشش عبارات إلى مستويين أو ثلاثة، ستعطيك المسافات البادئة المضاعفة أربعة أو ستة مستويات من المحاذاة. سيبدو التنسيق الناتج أكثر تعقيداً مما هي عليه الشفرة الفعلية. تجنب هذه المشكلة باستخدام محاكاة الكتلة النقية أو باستخدام *begin* و *end* كحدود الكتلة ومحاذاة *begin* و *end* مع العبارات التي تغلقان عليها.

اعتبارات أخرى

مع أن المسافات البادئة للكتل هي القضية الرئيسة في تنسيق بنى التحكم، ستلتقي بأنواع قليلة أخرى من القضايا، لذا إليك هنا بعض التوجيهات:

استخدم أسطراً فارغة بين الفقرات بعض الكتل من الشفرة حدودها غير محددة بأزواج *begin-end*. ينبغي أن تُعامل الكتلة المنطقية-مجموعة عبارات تنتمي إلى بعضها البعض-بالطريقة التي تُعامل بها الفقرات في الإنكليزية. افصلها عن بعضها البعض بأسطر فارغة. يظهر التعداد 31-29 مثالاً عن فقرات ينبغي أن تُفصل:

التعداد 31-29 مثال سي ++ عن شفرة ينبغي أن تُجمَّع وتُفصل.

```
cursor.start       = startingScanLine;
cursor.end         = endingScanLine;
window.title       = editWindow.title;
window.dimensions = editWindow.dimensions;
window.foregroundColor  = userPreferences.foregroundColor;
cursor.blinkRate        = editMode.blinkRate;
window.backgroundColor  = userPreferences.backgroundColor;
SaveCursor( cursor );
SetCursor( cursor );
```

تبدو هذه الشفرة تمام جيدة، لكن السطور الفارغة ستحسنها بطريقتين[1]. أولاً، عندما يكون لديك مجموعة من العبارات التي لا يجب أن تُنفَّذ بترتيب محدد، فهذه الطريقة هي محاولة لجمعها كلها مع بعضها. لا تحتاج إلى تنقية أكثر لترتيب العبارات من أجل الحاسوب، لكن القراء البشر يقدّرون الدلائل الإضافية عن أي من العبارات تحتاج أن تُنفّذ بطريقة محددة وأي من العبارات لا تهتم بالترتيب. الانضباط بوضع سطور فارغة على طول البرنامج يجعلك تفكر بجد أكثر عن أي العبارات تنتمي إلى بعضها. المقطع المنقح في التعداد 31-30 يُظهر كيف ينبغي أن تُنظم هذه التجميعة.

تعداد 31-30 مثال سي++ على الشفرة التي يتم تجميعها بشكل صحيح وفصلها.

```
window.dimensions = editWindow.dimensions;
window.title = editWindow.title;
window.backgroundColor = userPreferences.backgroundColor;
window.foregroundColor = userPreferences.foregroundColor;

cursor.start = startingScanLine;
cursor.end = endingScanLine;
cursor.blinkRate = editMode.blinkRate;
SaveCursor( cursor );
SetCursor( cursor );
```

تهيئ هذه الأسطر نافذة النص

تهيئ هذه الأسطر المؤشر "المزلقة" وينبغي أن تُفصل عن الأسطر

تُظهر الشفرة المعاد تنظيمها أن أمرين يحدثان. في المثال الأول، عوز تنظيم العبارات وعوز السطور الفارغة، وحيلة علامات المساواة المنتظمة القديمة، تجعل العبارات تبدو مرتبطة أكثر مما هي عليه.

[1] **إشارة مرجعية** إن استخدمت عملية البرمجة بالشفرة الزائفة، ستفصل كتل الشفرة تلقائياً. لتفاصيل، ألق نظرة على الفصل 9، "عملية البرمجة بالشفرة الزائفة."

الطريقة الثانية التي بها تميل السطور الفارغة لتحسين الشفرة هي أنها تفتح مسافات طبيعية للتعليقات. في التعداد 31-30، سيدعم تعليقٌ فوق كل كتلة بشكل رائع التنسيق المُحسَّن.

نسّق كتل العبارة الواحدة بشكل متماسك كتلة العبارة الواحدة هي عبارة واحدة لاحقة بتركيبة تحكم، كعبارة واحدة لاحقة باختبار *if*. في هكذا حالة، *begin* وend ليستا لازمتين لتصحيح الترجمة ولديك خيارات الأسلوب الثلاثة الظاهرة في التعداد 31-31

تعداد 31-31 مثال جافا لخيارات النمط لكتل مفردة العبارة.

```
if ( expression )                    الأسلوب 1
    one-statement;

if (expression ) {                   الأسلوب 2أ
    one-statement;
}

if (expression )                     الأسلوب 2ب
    {
    one-statement;
    }

if (expression )    one-statement;   الأسلوب 3
```

يوجد حجج لتفضيل كل واحد من هذه النهج. يتبع الأسلوب 1 مخطط المسافات البادئة المستخدم في الكتل، لذا فهو متماسك مع النهج الأخرى. الأسلوب 2 (سواء 2أ أو 2ب) هو أيضاً متماسك، وزوج *begin-end* يخفّض فرصة إضافة عبارات بعد *if* ونسيان إضافة *begin* و *end*. والذي سيكون خطأ مخادعاً بشكل خاص لأن المسافات البادئة تخبرك أن كل شيء على ما يرام، لكن المسافات البادئة لن تُفسَّر بنفس الطريقة من قبل المترجم. الميزة الرئيسة للأسلوب 3 على الأسلوب 2 هي أنه أسهل كتابةً. وميزته على الأسلوب 1 أنه إن نُسخ إلى مكان آخر في البرنامج، فإنه يمتلك فرصة أكبر ليُنسَخ بشكل صحيح. سيئته هي أنه في *المصحح* الموجهة بالسطر، يعامل *المصحح* السطر كسطر واحد ولا يظهر *المصحح* لك إن كان ينفذ العبارة بعد اختبار *if* أو لا. كنت أستخدم الأسلوب 1 وكنت ضحية للتعديلات غير الصحيحة مرات عديدة. أنا لا أحب الاستثناء في استراتيجية المسافات البادئة المسبب بالأسلوب 3، لذا فإنني أتجنبه بشكل كامل. في المشاريع ضمن مجموعة، أنا أفضل أي شكل من الأسلوب 2 لتماسكه وقابليته للتعديل الآمن. بغض النظر عن الأسلوب الذي تختار، استخدمه بشكل متماسك واستخدم نفس الأسلوب لاختبار *if* وكل الحلقات.

من أجل التعابير المعقدة، ضع الشروط المستقلة في أسطر مستقلة ضع كل جزء من التعبير المعقد في سطره الخاص. يُظهر التعداد 31-32 تعبيراً منسّقا بدون أي انتباه إلى قابلية القراءة:

التعداد 31-32 **مثال جافا عن تعبير معقد غير منسق جوهرياً (وغير مقروء).**[1]

```
if ((('0' <= inChar) && (inChar <= '9')) || (('a' <= inChar) &&
   (inChar <= 'z')) || (('A' <= inChar) && (inChar <= 'Z')))
   ...
```

هذا مثال عن التنسيق للحاسوب وليس للقراء البشر. بتكسير التعبير إلى عدة أسطر، كما في التعداد 31-33، تستطيع أن تُحسّن قابلية القراءة.

التعداد 31-33 **مثال جافا عن تعبير معقد مقروء.**

```
if ((('0' <= inChar) && (inChar <= '9')) ||
   (('a' <= inChar) && (inChar <= 'z')) ||
   (('A' <= inChar) && (inChar <= 'Z')))
   ...
```

تستخدم القطعة الثانية تقنيات تنسيق متعددة-المسافات البادئة والفراغات وترتيب رقم السطر وتوضيح كل سطر غير منته-والنتيجة هي تعبير مقروء. أكثر بعد، الغاية من الاختبار أصبحت واضحة. إن احتوى التعبير خطأ أصغرياً، كاستخدام *z* بدلاً من *Z*، سيكون واضحاً في شفرة منسَّقة بهذه الطريقة، بينما لن يكون الخطأ واضحاً في تنسيق أقل حذر.

***تجنب* gotos**[2] كان السبب الأصلي وراء تَجنُّب *gotos* هو أنها جعلت إثبات صحة برنامج أمراً صعباً. إنها حجة مليحة لكل الناس الذين يريدون أن يثبتوا صحة برامجهم. والذين هم بالواقع لا أحد. المشكلة الضاغطة أكثر على معظم المبرمجين هي أن *gotos* تجعل الشفرة صعبة التنسيق. هل تضع مسافات بادئة أمام كل الشفرة بين *goto* والعلامة التي تذهب إليها؟ ماذا لو كان لديك عدة *gotos* إلى نفس العلامة؟ هل تُنظِّم كل واحدة جديدة تحت التي تسبقها؟ هاهنا بعض النصائح لتنسيق *gotos*:

- تجنب gotos. هذا يتجاوز مشكلة التنسيق كلياً.[3]
- استخدم اسماً كل حروفه كبيرة للعلامة التي تذهب إليها الشفرة. هذا يجعل العلامة واضحة.
- ضع العبارة الحاوية على goto في سطر بمفردها. هذا يجعل goto واضحة.

---

[1] **إشارة مرجعية** وضع التعابير المعقدة في توابع منطقية هو تقنية أخرى لجعل التعابير المعقدة مقروءة. لتفاصيل حول هذه التقنية وتقنيات تسهيل القراءة الأخرى، انظر القسم 19. 1، "التعابير المنطقية".

[2] **إشارة مرجعية** لتفاصيل حول استخدام *gotos*، انظر القسم 17. 3، "*goto*".

[3] ينبغي أن تكون عبارات *goto* ذات محاذاة يسارية وينبغي أن تتضمن اسم المبرمج وهاتف بيته ورقم بطاقته الائتمانية. --عبدول نزار

- ضع العلامة التي تذهب إليها goto في سطر لوحدها. أحطها بسطور فارغة. هذا يجعل العلامة واضحة. خذ مســافات بادئة من أمام الســطر الحاوي على العلامة ليصــل إلى الحافة اليســرى لتجعل العلامة واضحة بقدر الإمكان.

التعداد 31-34 يُظهر أعراف تنسيق goto بالعمل:

التعداد 31-34 ***مثال سي++ عن القيام بأفضل ما يمكن لحالة سيئة (استخدام goto).***[1]

```
void PurgeFiles( ErrorCode & errorCode ) {
   FileList fileList;
   int numFilesToPurge = 0;
   MakePurgeFileList( fileList, numFilesToPurge );

   errorCode = FileError_Success;
   int fileIndex = 0;
   while ( fileIndex < numFilesToPurge ) {
      DataFile fileToPurge;
      if ( !FindFile( fileList[ fileIndex ], fileToPurge ) ) {
         errorCode = FileError_NotFound;
         goto END_PROC;
      }

      if ( !OpenFile( fileToPurge ) ) {
         errorCode = FileError_NotOpen;
         goto END_PROC;
      }

      if ( !OverwriteFile( fileToPurge ) ) {
         errorCode = FileError_CantOverwrite;
         goto END_PROC;
      }

      if ( !Erase( fileToPurge ) ) {
         errorCode = FileError_CantErase;
         goto END_PROC;
      }
      fileIndex++;
   }

END_PROC:

   DeletePurgeFileList( fileList, numFilesToPurg
}
```


مثال **سي** ++ في التعداد 31-34 طويل نسبياً لذا تستطيع أن تجد حالة يقرر فيها مبرمج خبير وهو واعٍ أن *goto* هي خيار التصميم الأفضل.[2] في هكذا حالة، التنسيق الظاهر هو تقريباً أفضل شيء يمكنك فعله.

لا استثناءات في سطر النهاية من أجل عبارات **case** إحدى المجازفات لتنسيق سطر النهاية تظهر في تنسيق عبارات *case*. أسلوب شائع لتنسيق الحالات هو أن تسبقها بمسافات بادئة لتصل إلى يمين توصيف كل حالة، كما هو ظاهر في التعداد 31-35. المشكلة الكبيرة في هذا الأسلوب هي أنه وجع رأس في الصيانة.

[1] إشارة مرجعية لطرق أخرى لمعالجة هذه المشكلة، انظر "معالجة الأخطاء وgotos" في القسم 17. 3.

[2] **إشارة مرجعية** لتفاصيل عن استخدام عبارات *case* الق نظرة على القسم 15. 2، "عبارات *case*".

التعداد 31-35 مثال سي ++ عن تنسيق سطر النهاية صعب الصيانة لعبارة case.

```
switch ( ballColor ) {
   case BallColor_Blue:                Rollout();
                                       break;
   case BallColor_Orange:              SpinOnFinger();
                                       break;
   case BallColor_FluorescentGreen: Spike();
                                       break;
   case BallColor_White:               KnockCoverOff();
                                       break;
   case BallColor_WhiteAndBlue:        if ( mainColor == BallColor_White ) {
                                          KnockCoverOff();
                                       }
                                       else if ( mainColor == BallColor_Blue ) {
                                          RollOut();
                                       }
                                       break;
   default:                            FatalError( "Unrecognized kind of ball." );
                                       break;
}
```

إن أضفت حالة باسم أطول من أي من الأسماء الموجودة، فعليك أن تزيح كل الحالات والشفرات الموافقة لها. تجعل المسافة البادئة الضخمة من ملائمة أي منطق إضافي أمراً صعباً، كما هو ظاهر في حالة *WhiteAndBlue*. الحل هو أن تبدّل إلى زيادتك المعيارية للمسافات البادئة. إن كنت تضع مسافات بادئة من ثلاثة فراغات للعبارات في حلقة، ضع مسافات بادئة بنفس عدد الفراغات للحالات في عبارة *case*، كما في التعداد 31-36:

التعداد 31-36 مثال سي ++ عن مسافات بادئة معيارية جيدة لعبارة case.

```
switch ( ballColor ) {
   case BallColor_Blue:
      Rollout();
      break;
   case BallColor_Orange:
      SpinOnFinger();
      break;
   case BallColor_FluorescentGreen:
      Spike();
      break;
   case BallColor_White:
      KnockCoverOff();
      break;
   case BallColor_WhiteAndBlue:
      if ( mainColor == BallColor_White ) {
         KnockCoverOff();
      }
      else if ( mainColor == BallColor_Blue ) {
         RollOut();
      }
      break;
   default:
      FatalError( "Unrecognized kind of ball." );
      break;
}
```

هذه حالة فيها قد يفضل العديد من الناس منظر المثال الأول. بالنظر إلى إمكانية ملائمة سطور أطول والتماسك وقابلية الصيانة، على كل، يفوز النهج الثاني بكل سهولة.

إن كان لديك عبارة *case* فيها كل الحالات متوازية بالضبط وكل الأفعال قصيرة، قد تفكر بوضع الحالة والفعل على نفس السطر. في معظم الأمثلة، على كل، ستعيش لتندم على ذلك. التنسيق هو ألم في البداية ويتكسّر تحت التعديلات، ومن الصعب الحفاظ على هيكلة كل الحالات متوازية طالما أن بعض من الأفعال القصيرة تصبح أطول.

31. 5 رسم العبارات المفردة

يشرح هذا القسم طرقاً عديدة لتحسين العبارات المفردة في برنامج.

طول العبارة

تحديد طول سطر العبارة ب 80 محرف هو قاعدة شائعة وربما منتهية الصلاحية[1]. إليك الأسباب:

- السطور الأطول من 80 محرف صعبة القراءة.
- حد ال 80 محرف يثبط التعشيش العميق.
- السطور الأطول من 80 محرف غالباً لا تتناسب مع الأوراق 8.5" × 11"، وخصوصاً عندما تُطبع الشفرة "2 فوق" (صفحتين من الشفرة على كل صفحة طباعة بالواقع).

مع شاشات أكبر وخطوط ذات محارف أقل سماكة ونظام المنظر الطبيعي (`landscape`) يبدو حد ال 80 محرف اعتباطياً. سطر مفرد طوله 90 محرفاً عادةً أكثر قابلية للقراءة من واحد قُسّم إلى اثنين فقط لتجنب تجاوز العمود الثمانين. مع التقنيات الحديثة، أحياناً يكون تجاوز ال 80 عمود أمراً حسناً.

استخدام الفراغات من أجل الوضوح

أضف مسافات بيضاء ضمن العبارة لصالح قابلية القراءة:

استخدم فراغات لتجعل التعابير المنطقية مقروءة   التعبير

```
while(pathname[startPath+position]<>';') and
    ((startPath+position)<length(pathname)) do
```

مقروء تقريباً بمقدار أتحداكأنتقرأهذا `Idareyoutoreadthis`.

[1] **إشارة مرجعية** لتفاصيل عن توثيق العبارات المفردة، الق نظرة على "التعليق على السطور المفردة" في القسم 32. 5.

كقاعدة، ينبغي أن تفصل المُعرِّفات عن المعرفات الأخرى بالفراغات. إن استخدمت هذه القاعدة، سيبدو تعبير *while* كالتالي:

```
while ( pathname[ startPath+position ] <> ‘;’ ) and
    (( startPath + position ) < length( pathname )) do
```

قد ينصح بعض الفنانين البرمجيين بتحسين هذا التعبير الخاص بفراغات إضافية للتأكيد على بنيانه المنطقي، بهذه الطريقة:

```
while ( pathname[ startPath + position ] <> ‘;’ ) and
    ( ( startPath + position ) < length( pathname ) ) do
```

هذا أمر حسن، مع أن الاستخدام الأول للفراغات كان كافياً لضمان قابلية القراءة. من الصعب جداً أن تسبب الفراغات الزائدة الأذى، على كل، لذا كن سخياً بها.

استخدم فراغات لتجعل ما تدل عليه المصفوفات مقروءاً التعبير

```
grossRate[census[groupId].gender,census[groupId].ageGroup]
```

ليس مقروءاً أكثر من تعبير *while* الثقيل السابق. استخدم فراغات حول كل فهرس في المصفوفة لتجعل الفهارس مقروءة. إن استخدمت هذه القاعدة، سيبدو التعبير كالتالي:

```
grossRate[ census[ groupId ].gender, census[ groupId ].ageGroup ]
```

استخدم فراغات لتجعل وسطاء الإجرائيات مقروءة ما هو الوسيط الرابع للإجرائية التالية؟

```
ReadEmployeeData(maxEmps,empData,inputFile,empCount,inputError);
```

الآن، ما هو الوسيط الرابع للإجرائية التالية؟

```
GetCensus( inputFile, empCount, empData, maxEmps, inputError );
```

أي الاثنين كان إيجاده أسهل؟ هذا سؤال واقعي وجدير بالاهتمام لأن مواقع الوسطاء ذات شأن في كل اللغات الإجرائية الرائدة. أن يكون لديك توصيف لإجرائية في أحد نصفي الشاشة واستدعاء هذه الإجرائية في النصف الثاني، ولتقارن كل وسيط رسمي بكل وسيط فعلي.

تنسيق سطور الإكمال

إحدى المشاكل الأكثر إزعاجاً في مجال تنسيق البرنامج هي اتخاذ قرار بمَ ستفعل مع جزء العبارة الذي يفيض إلى السطر التالي. هل تضع مسافات بادئة أمامه بنفس مقدار المسافات البادئة العادية؟ هل تحاذيه تحت الكلمة المفتاحية؟ ماذا عن الإسنادات؟

هاهنا نهج متماسك ذو معنى ومفيد بشكل خاص في **جافا وسي وسي++ وفيجوال بيسك**، واللغات الأخرى التي تحث على استخدام أسماء متحولات طويلة:

اجعل نقصان العبارة واضحاً. أحياناً يتحتم على العبارة أن تنقسم عبر الأسطر، إما لأنها أطول مما تسمح معايير البرمجة أو لأنها طويلة بشكل غريب جداً حتى توضع في سطر واحد. وضّح أن قسم العبارة في السطر الأول هو فقط قسم من العبارة. الطريقة الأسهل لتفعل ذلك هي أن تُجزّئ العبارة بحيث يكون الجزء في السطر الأول غير صحيح قواعدياً إن قُيِّم بمفرده. بعض الأمثلة ظاهرة في التعداد 31-37:

التعداد 31-37 **أمثلة بلغة جافا عن عبارات غير مكتملة بشكل واضح.**

```
while ( pathName[ startPath + position ] != ';' ) &&
   ( ( startPath + position ) <= pathName.length() )
...

totalBill = totalBill + customerPurchases[ customerID } +
   SalesTax( customerPurchases[ customerID ] );
...

DrawLine( window.north, window.south, window.east, window.west
   currentWidth, currentAttribute );
...
```

&& تشير إلى أن العبارة غير منتهية

علامة الجمع (+) تشير إلى أن العبارة غير منتهية

الفاصلة (,) تشير إلى أن العبارة غير منتهية

بالإضافة إلى إخبار القارئ أن العبارات غير منتهية في السطر الأول، يساعد التجزيء بمنع التعديلات غير الصحيحة. إن حُذف إكمال العبارة، لن يبدو السطر الأول وكأنك نسيت قوساً أو فاصلة منقوطة فقط-إنها تحتاج بوضوح شيئاً ما إضافياً.

نهج بديل يعمل بشكل جيد هو أن تضع محرف الإكمال في بداية سطر الإكمال، كما هو ظاهر في التعداد 31-38.

التعداد 31-38 **أمثلة بلغة جافا عن عبارات غير منتهية بشكل واضح-أسلوب بديل.**

```
while ( pathName[ startPath + position ] != ';' )
   && ( ( startPath + position ) <= pathName.length() )
...

totalBill = totalBill + customerPurchases[ customerID }
   + SalesTax( customerPurchases[ customerID ] );
...
```

مع أن هذا الأسلوب لن ينتج خطأً قواعدياً على *&&* أو *+* المعلقة، إنه يجعل فحص العمليات في الحافة اليسرى من العمود، في المكان الذي ينتظم فيه النص، أسهل منه في الحافة اليمنى، في المكان الذي يكون النص أشعثاً. إنه يمتلك الحسنة الإضافية لتوضيح بنيان العمليات، كما هو ظاهر في التعداد 31-39.

التعداد 31-39 **مثال جافا عن أسلوب يوضح العمليات المعقدة.**

```
totalBill = totalBill
   + customerPurchases[ customerID ]
   + CitySalesTax( customerPurchases[ customerID ] )
   + StateSalesTax( customerPurchases[ customerID ] )
   + FootballStadiumTax()
   - SalesTaxExemption( customerPurchases[ customerID ] );
```

ابق العناصر المترابطة جداً مع بعضها البعض عندما تُجزِّء سطراً، حافظ على العناصر التي تنتمي إلى بعضها البعض مجتمعة: إشارات المصفوفات، والوسطاء إلى الإجرائية، وهلم جراً. المثال الظاهر في التعداد 31-40 هو صيغة رديئة:

التعداد 31-40 **مثال جافا عن تجزيء سطر برداءة.**

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) + LateCharge(
   paymentHistory[ customerID ] );
```

على نحو لا يمكن إنكاره، تجزئة السطر هذه تتبع توجيه جعل النقص في العبارة واضحاً، لكنها تفعل ذلك بطريقة تجعل العبارة صعبة القراءة من دون ضرورة لذلك. قد تجد حالة تكون فيها التجزئة ضرورية، لكن في هذه الحالة ليس الأمر كذلك. من الأفضل المحافظة على إشارات المصفوفة كلها على سطر واحد. يظهر التعداد 31-41 تنسيقاً أفضل:

التعداد 31-41 **مثال جافا عن تجزيء سطر بشكل جيد.**

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) +
   LateCharge( paymentHistory[ customerID ] );
```

ضع مسافات بادئة أمام سطور إكمال استدعاءات الإجرائيات بالمقدار المعياري إن كنت تضع عادةً مسافات بادئة بثلاثة فراغات للعبارات في الحلقات أو الشرطيات، ضع مسافات بادئة لسطور تكملة الإجرائيات بثلاثة فراغات. بعض الأمثلة ظاهرة في التعداد 31-42:

التعداد 31-42 **أمثلة بلغة جافا عن وضع مسافات بادئة أمام سطور تكملة استدعاء إجرائية باستخدام المقدار المعياري للمسافات البادئة.**

```
DrawLine( window.north, window.south, window.east, window.west,
   currentWidth, currentAttribute );
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],
   italic[ fontId ], syntheticAttribute[ fontId ].underline,
   syntheticAttribute[ fontId ].strikeout );
```

أحد البدائل لهذا النهج هو أن تُنظِّم سطور الإكمال تحت أول وسيط للإجرائية، كما هو ظاهر في التعداد 31-43:

التعداد 31-43 **أمثلة جافا عن وضع مسافات بادئة لسطر إكمال استدعاء إجرائية للتأكيد على أسماء الإجرائيات.**

```
DrawLine( window.north, window.south, window.east, window.west,
          currentWidth, currentAttribute );
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],
                   italic[ fontId ], syntheticAttribute[ fontId ].underline,
                   syntheticAttribute[ fontId ].strikeout );
```

من الناحية الجمالية، يبدو هذا أشعثاً قليلاً بالمقارنة مع النهج الأول. إنه أيضاً صعب الصيانة عند تغيّر أسماء الإجرائية، وتغير أسماء الوسطاء، وهلم جراً. يميل معظم المبرمجين إلى الانجذاب باتجاه الأسلوب الأول مع مرور الوقت.

اجعل إيجاد نهاية سطور الإكمال أمراً سهلاً إحدى المشاكل مع النهج الظاهر فوق أنك لا تستطيع بسهولة أن تجد نهاية كل سطر. بديل آخر هو أن تضع كل وسيط على سطر لوحده وتحدد نهاية المجموعة بقوس الإغلاق. يُظهر التعداد 31-44 كيف يبدو.

التعداد 31-44 أمثلة جافا عن تنسيق سطور إكمال استدعاء الإجرائيات، وسيط واحد على سطر واحد.

```
DrawLine(
   window.north,
   window.south,
   window.east,
   window.west,
   currentWidth,
   currentAttribute
);

SetFontAttributes(
   faceName[ fontId ],
   size[ fontId ],
   bold[ fontId ],
   italic[ fontId ],
   syntheticAttribute[ fontId ].underline,
   syntheticAttribute[ fontId ].strikeout
);
```

كما هو واضح، يستهلك هذا النهج الكثير من المساحة الحقيقية. إن كانت وسطاء الإجرائية مراجع لحقل كائن طويل أو أسماء مؤشرات، على كل، كما هما الوسيطين الآخِرين، يحسّن استخدام وسيط واحد لكل سطر قابلية القراءة بشكل جوهري. تجعل ال); في نهاية الكتلة نهاية الاستدعاء واضحة. ولا يجب عليك أن تعيد التنسيق عندما تضيف وسيطاً؛ فقط تضيف سطراً جديداً.

في التطبيق، عادةً قلة فقط من الإجرائيات تحتاج أن تُجزأ إلى عدة سطور. تستطيع أن تتعامل مع الأخرى بسطر واحد. أي من الخيارات الثلاثة لتنسيق السطور المتعددة لاستدعاءات الإجرائية تعمل على ما يرام إن استخدمته بثبات.

ضع مسافات بادئة أمام سطور إكمال عبارات التحكم بالمقدار المعياري إن نفذت لديك المساحة من حلقة *for* أو حلقة *while* أو عبارة *if*، فقم بوضع مسافات بادئة أمام سطر الإكمال بنفس مقدار الفراغات الذي تُبدئ به العبارات في حلقة أو بعد عبارة *if*. مثالين ظاهرين في التعداد 31-45:

التعداد 31-45 مثالي **جافا** عن وضع مسافات بادئة أمام سطور إكمال عبارات التحكم.

```
while ( ( pathName[ startPath + position ] != ';' ) &&
   ( ( startPath + position ) <= pathName.length() ) ) {
   ...
}

for ( int employeeNum = employee.first + employee.offset;
   employeeNum < employee.first + employee.offset + employee.total;
   employeeNum++ ) {
   ...
}
```

سطر الإكمال هذا مسبوق بمسافات بادئة بالعدد المعياري للفراغات

وكذلك هذين

هذا يلاقي مجموعة المعايير المذكورة سابقاً في هذا الفصل.[1] جزء الإكمال لهذه العبارة تمَّ بشكل منطقي-إنها دائما تنتظم تحت العبارة التي تُكملها. يمكن أن ينجز الإكمال بشكل متماسك-إنه يستخدم فقط عدة فراغات زيادة على السطر الأصلي. إنه مقروء كأي شيء آخر، وهو بمقدار قابلية صيانة أي شيء آخر. في بعض الحالات قد تكون قادراً على تطوير قابلية القراءة عن طريق معايرة طفيفة للمسافات البادئة أو لوضع الفراغات، لكن تأكد أن تحافظ على مقايضة قابلية الصيانة في عقلك عندما تفكر بالمعايرة الطفيفة.

لا تحاذِ الحوافّ اليمنى لعبارات الإسناد لقد نصحت في النسخة الأولى من هذا الكتاب بمحاذاة الحواف اليمنى للعبارات الحاوية على إسناد كما هو ظاهر في التعداد 31-46:

التعداد 31-46 مثال جافا عن تنسيق سطر النهاية مستخدَم لإكمال عبارة إسناد-تطبيق سيء.

```
customerPurchases   = customerPurchases + CustomerSales( CustomerID );
customerBill        = customerBill + customerPurchases;
totalCustomerBill   = customerBill + PreviousBalance( customerID ) +
                      LateCharge( customerID );
customerRating      = Rating( customerID, totalCustomerBill );
```

وجدت بفضل فهم 10 سنوات مضت أنه، بينما يبدو أسلوب المسافات البادئة هذا جذاباً، يصبح وجع رأس عند الحفاظ على محاذاة إشارات المساواة بينما تتغير أسماء المتحولات وتشغيل الشفرة عبر أدوات تُبدل الفراغات بعلامات الجدولة وعلامات الجدولة بالفراغات. إنه صعب الصيانة أيضاً عندما تنتقل السطور بين أقسام مختلفة من البرنامج ذات مستويات مختلفة للمحاذاة.

[1] **إشارة مرجعية** في بعض الأحيان الحل الأمثل لاختبار معقد هو أن تضعه في تابع منطقي. لأمثلة، الق نظرة على "جعل التعابير المعقدة بسيطة" في القسم 19 .1.

من أجل التماسك مع توجيهات المحاذاة الأخرى بالإضافة لقابلية الصيانة، عامل مجموعات العبارات الحاوية على عمليات إسناد تماماً كما تعامل العبارات الأخرى، كما يُظهر التعداد 31-47:

التعداد 31-47 مثال جافا عن محاذاة معيارية لإكمال عبارة إسناد-تطبيق جيد.

```
customerPurchases = customerPurchases + CustomerSales( CustomerID );
customerBill = customerBill + customerPurchases;
totalCustomerBill = customerBill + PreviousBalance( customerID ) +
   LateCharge( customerID );
customerRating = Rating( customerID, totalCustomerBill );
```

ضع مسافات بادئة أمام سطور إكمال عبارات الإسناد بالمقدار المعياري في التعداد 31-47 سطر الإكمال لعبارة الإسناد الثالثة وُضع له المقدار المعياري للمسافات البادئة. لقد أُنجز هذا الأمر لنفس أسباب أن عبارات الإسناد بالعموم لا تُنسَّق بأي طريقة خاصة: قابلية قراءة وصيانة عامة.

استخدام عبارة واحدة في كل سطر

تسمح لك اللغات الحديثة مثل **سي ++ وجافا** بوضع عدة عبارات على السطر الواحد. قوة التنسيق الحر هو بَرَكة مختلطة، على كل، عندما يحدث وتضع عدة عبارات على سطر. سيحتوي هذا السطر عدة عبارات يمكن أن تُفصل بشكل منطقي إلى عدة سطور مستقلة.

```
I = 0; j = 0; k = 0; DestroyBadLoopNames( I, j, k );
```

إحدى الحجج في تفضيل عبارة واحدة على عدة عبارات توضع على سطر واحد هي أنها تتطلب سطوراً أقل من مساحة الشاشة أو ورقة الطابعة، ما يسمح لمزيد من الشفرة أن تظهر معاً. إنها أيضاً طريقة لتجميع العبارات المترابطة، ويؤمن بعض المبرمجين بأنها تقدّم دلائل للمعايرة إلى المترجم.

تلك أسباب جيدة، لكن أسباب أن تحدّ نفسك بعبارة واحدة في السطر الواحد أكثر قوة:

- يقدم وضــع كل عبارة على ســطر بمفردها نظرة دقيقة عن تعقيد البرنامج. إنه لا يُخفي التعقيد بجعل العبارات المعقدة تبدو سخيفة. فالعبارات المعقدة تبدو معقدة. والعبارات السهلة تبدو سهلة.
- وضع عدة عبارات على سطر واحد لا يقدّم دلائل للمعايرة إلى المترجمات الحديثة.[1] لا تعتمد مترجمات التحسين التي تظهر هذه الأيام على دلائل التنسيق لتقوم بمعايراتها. هذا مشروح لاحقاً في هذا القسم.
- بعبارات على سطورها الخاصة، تُقرأ الشفرة من الأعلى إلى الأسفل، وليس من الأعلى إلى الأسفل ومن اليســار إلى اليمين. عندما تنظر إلى ســطر محدد من الشــفرة، ينبغي أن تكون عيناك قادرتان على تتبع

[1] **إشارة مرجعية** تمت مناقشة تحسينات أداء مستوى الشفرة في الفصل 25، "استراتيجيات ضبط الشفرة"، والفصل 26، "تقنيات ضبط الشفرة".

الحافة اليسرى للشفرة. لا ينبغي أن تُجبَر عيناك على الغوص في كل سطر فقط لأن سطراً مفرداً قد يحتوي على عبارتين.

- مع عبارات على سطورها الخاصة، يكون إيجاد الأخطاء القواعدية سهلاً عندما يقدم مترجمك أرقام سطور الأخطاء فقط. إن كان لديك عدة عبارات على سطر، لا يخبرك رقم السطر بالعبارة التي تحتوي على خطأ.
- مع عبارة واحدة على سطر، يكون من السهل أن تخطو عبر الشفرة بـ*مصححات* موجهة بالسطر. إن كان لديك عدة عبارات على سطر، ستنفذها *المصححات* كلها دفعة واحدة وسيكون عليك أن تنتقل إلى المجمع لتخطو عبر العبارات المفردة.
- بواحدة على سطر، يكون من السهل تحرير العبارات المفردة-بحذف سطر أو تحويله مؤقتاً إلى تعليق. إن كان لديك عدة عبارات على سطر، فعليك أن تقوم بتحريرك بين العبارات الأخرى.

في سي ++ تَجنَّب استخدام عدة عمليات على السطر الواحد (الآثار جانبية) الآثار الجانبية هي نتائج لعبارة مختلفة عن نتيجتها الرئيسة. في **سي ++**، وجود المعامل ++ على سطر يحتوي على عمليات أخرى هو أثر جانبي. كذلك، إسناد قيمة إلى متحول واستخدام الجهة اليسرى من الإسناد في شرطية هو أثر جانبي.

تتجه الآثار الجانبية لجعل الشفرة صعبة القراءة. مثالاً، إن كان n يساوي 4، ما هو خرج العبارة الظاهرة في التعداد 31-48؟

التعداد 31-48 **مثال سي ++ أثر جانبي لا يمكن التكهن به.**

```
printMessge( ++n, n + 2 );
```

هل 4 و6؟ أم 5 و7؟ أم 5 و6؟ الجواب هو "ليس أياً مما سبق." الوسيط الأول، *++n*، هو 5. لكن لغة **سي ++** لا تُعرِّف الترتيب الذي به تُقيَّم الحدود في تعبير أو الوسطاء إلى إجرائية. لذا يمكن أن يُقيِّم المترجم الوسيط الثاني، *n + 2*، قبل أو بعد الوسيط الأول، قد تكون النتيجة 6 أو 7، بالاعتماد على المترجم. يُظهر التعداد 31-49 كيف ينبغي أن تكتب العبارة بحيث يكون القصد واضحاً:

التعداد 31-49 **مثال سي ++ عن اجتناب أثر جانبي غير قابل للتكهن.**

```
++n;
printMessge( n, n + 2 );
```

إن كنت لا تزال غير مقتنعاً أنه ينبغي عليك أن تضع الآثار الجانبية على سطور لوحدها، حاول أن تكتشف ما الذي تفعله الإجرائية الظاهرة بالتعداد 31-50:

التعداد 31-50 **مثال سي عن العديد من العمليات على سطر.**

```
strcpy( char * t, char * s ) {
   while ( *++t = *++s )
      ;
}
```

بعض مبرمجي **سي** لا يرون التعقيد في هذا المثال لأنه تابع شائع. إنهم ينظرون إليه ويقولون، "هذا هو *strcpy()*." في هذه الحالة، على كل، إنه ليس تماماً *strcpyu()*. إنه يحتوي على خطأ. إن قلت، "هذا هو *strcpy()*" عندما رأيت الشفرة، فإنك كنت تتعرّف على الشفرة ولا تقرؤها. هذه هي تماماً الحالة التي تكون بها عندما *تصحح* برنامجاً: الشفرة التي تُغفلها لأنك" تتعرّف" عليها بدلاً من أن تقرأها يمكن أن تحتوي على خطأ أصعب في الاكتشاف مما يلزم أن يكون.

القطعة الظاهرة في التعداد 31-51 هي مماثلة وظيفياً للأولى وأكثر قابلية للقراءة:

التعداد 31-51 مثال سي عن عدة عمليات مقروءة كل واحدة على سطر.

```
strcpy( char * t, char * s ) {
   do {
      ++t;
      ++s;
      *t = *s;
    }
   while ( *t != '\0' );
}
```

في الشفرة المعاد تنسيقها، الخطأ بيّن. بشكل واضح، تتم زيادة *t* و *s* قبل أن تُنسخ **s إلى *t*. المحرف الأول يسقط من النسخ.

يبدو المثال الثاني مفصلاً أكثر من الأول، حتى وإن كانت العمليات المنجزة في المثال الثاني والأول متطابقة. السبب في أنه يبدو أكثر تفصيلاً هو أنه لا يخفي تعقيد عملياته.

لا يسوغ الأداء المُحسَّن وضع عدة عمليات على نفس السطر أيضاً[1]. لأن إجرائيتي strcpy() متعادلتين منطقياً، فإنك تتوقع أن يولد المترجم شفرتين متطابقتين من كل منهما. عندما رُسم أداء إصداري الإجرائية، على كل، استغرقت الأولى 4.81 ثانية لنسخ 5000000 سلسلة نصية واستغرقت الثانية 4.35 ثانية.

في هذه الحالة، تسببت النسخة "الذكية" بضريبة سرعة مقدارها 11 بالمئة، ما يجعلها تبدو أقل ذكاء بكثير. تتغير النتائج من مترجم إلى مترجم، لكنها بالعموم تشير بأنه إلى أن تقيس مكاسب في الأداء، من الأفضل لك أن تكافح من أجل الوضوح والصحة أولاً، فالأداء ثانياً.

[1] **إشارة مرجعية** لتفاصيل عن معايرة الشفرة، راجع الفصل 25، "استراتيجيات معايرة الشفرة،" والفصل 26، "تقنيات معايرة الشفرة."

حتى وإن كنت تقرأ عبارات لها آثار جانبية بسهولة. اشعر بالرحمة تجاه الناس الآخرين الذين سيقرؤون شفرتك. يحتاج معظم المبرمجين الجيدين إلى التفكير مرتين كي يفهموا التعابير التي لها آثار جانبية. دعهم يستخدمون خلايا أدمغتهم ليفهموا أسئلة أكبر عن كيفية عمل شفرتك بدلاً من التفاصيل القواعدية لتعبير محدد.

رسم التصريحات عن البيانات

استخدم تصريحاً واحداً عن البيانات في السطر الواحد[1] كما يظهر في الأمثلة السابقة، ينبغي أن تعطي كل تصريح عن البيانات سطره الخاص. يكون وضع تعليق بجانب كل تصريح أسهل إن كان كل واحد على سطره الخاص. ويكون تعديل التصريحات أسهل لأن كل تصريح مستقل. ويكون إيجاد متحولات محددة أسهل لأنك تستطيع فحص عمود واحد بدلاً من قراءة كل سطر. ويكون إيجاد وإصلاح الأخطاء القواعدية أسهل لأن رقم السطر الذي يعطيكه المترجم فيه تصريح واحد فقط.

بسرعة-في التصريح عن البيانات في التعداد 31-52، ما هو نمط المتحول currentBottom؟

التعداد 31-52 مثال سي ++ عن ازدحام أكثر من تصريح عن متحول واحد على سطر.




```
int rowIndex, columnIdx; Color previousColor, currentColor, nextColor; Point
previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom; Font
previousTypeface, currentTypeface, nextTypeface; Color choices[ NUM_COLORS ];
```

هذا مثال متطرف، لكنه ليس مستبعداً جداً أن يكون قد أتى من الأسلوب الشائع أكثر بكثير الظاهر في التعداد 31-53:

التعداد 31-53 مثال سي ++ عن ازدحام أكثر من تصريح عن متحول واحد على سطر.




```
int rowIndex, columnIdx;
Color previousColor, currentColor, nextColor;
Point previousTop, previousBottom, currentTop, currentBottom, nextTop,
nextBottom;
Font previousTypeface, currentTypeface, nextTypeface;
Color choices[ NUM_COLORS ];
```

هذا ليس أسلوباً غير شائع في التصريح عن المتحولات، ولا يزال المتحول صعب الإيجاد لأن كل التصريحات مُكدَّسة مع بعضها. نمط المتحول صعب الإيجاد، أيضاً. الآن، ما هو نمط *nextColor* في التعداد 31-54؟

التعداد 31-54 مثال سي ++ عن قابلية قراءة مُنجَزة بوضع تصريح متحول واحد فقط على كل سطر.

[1] **إشارة مرجعية** لتفاصيل عن توثيق التصريحات عن البيانات، راجع " التعليق على التصريحات عن البيانات" في القسم 32. 5. لجوانب استخدام البيانات، انظر الفصول 10 إلى 13.

```
int rowIndex;
int columnIdx;
Color previousColor;
Color currentColor;
Color nextColor;
Point previousTop;
Point previousBottom;
Point currentTop;
Point currentBottom;
Point nextTop;
Point nextBottom;
Font previousTypeface;
Font currentTypeface;
Font nextTypeface;
Color choices[ NUM_COLORS ];
```

ربما كان المتحول *nextColor* أسهل إيجاداً من *nextTypeFace* في التعداد 31-53. هذا الأسلوب مميز بتصريح واحد للسطر الواحد وبتصريح كامل، متضمناً نمط المتحول، على كل سطر.

على نحو لا يمكن إنكاره، يأكل هذا الأسلوب الكثير من مساحة الشاشة-20 سطر بدلاً من ثلاثة في المثال الأول، مع أن تلك السطور الثلاثة كانت بشعة حقاً. لا أستطيع أن أشير إلى أي دراسة تُظهر أن هذا الأسلوب يقود إلى ثغرات أقل أو فهم أفضل. إن طلبت سالي المبرمج، الصغيرة، أن أراجع شفرتها، على كل، وكانت التصريحات عن البيانات تبدو مثل المثال الأول، سأقول "غير ممكن-إنها صعبة القراءة جداً." إن بدت مثل المثال الثاني، سأقول "أها...قد أعيدها إليك." إن بدت مثل المثال الأخير، سأقول "بالتأكيد-إنها متعة."

صرح عن المتحولات بالقرب من استخدامها الأول أن تصرح عن كل متحول بالقرب من مكان استخدامه الأول هو أسلوب مُفضَّل على التصريح عن كل المتحولات في كتلة كبيرة. هذا الأسلوب المفضَّل يُخفِّض "الامتداد" و"فترة الحياة" ويساعد على إعادة تصنيع الشفرة إلى إجرائيات أصغر عند الضرورة. لتفاصيل، راجع "حافظ على ' الحياة' قصيرة الأمد بقدر الإمكان" في القسم 10 .4.

رتّب التصريحات بشكل ذي معنى في التعداد 31-54، جُمّعت التصريحات وفق النمط. التجميع وفق النمط ذو معنى عادةً مذ أن المتحولات ذات النمط نفسه تتجه لكي تُستخدم في عمليات مترابطة. في الحالات الأخرى، قد تختار أن ترتبها بشكل أبجدي وفق اسم المتحول. مع أن الترتيب الأبجدي لديه محامون كثر، فإن إحساسي أنه يحتاج عملاً كثيراً جداً تجاه قيمته الفعلية. إن كانت لائحة المتحولات طويلة جداً بحيث يكون الترتيب الأبجدي مفيداً، فقد تكون إجرائيتك كبيرة جداً. جزّئها بحيث يكون لديك إجرائيات أصغر مع متحولات أقل.

في سي ++، ضع النجمة بجانب اسم المتحول في التصريح عن المؤشرات أو صرح عن أنماط للمؤشرات أن ترى تصريحات عن مؤشرات تضع النجمة بجانب النمط أمر شائع، كما في التعداد 31-55:

التعداد 31-55 **مثال سي ++ عن النجمة في التصريحات عن المؤشرات.**

```
EmployeeList* employees;
File* inputFile;
```

المشكلة في وضع النجمة بجانب النمط بدلا من اسم المتحول هي أنه عندما تضع أكثر من تصريح واحد على السطر، ستطبق النجمة فقط على المتحول الأول حتى وإن كان التنسيق المرئي يوحي أنها تُطبَق على كل المتحولات التي على السطر. تستطيع أن تتجنب هذه المشكلة بوضع النجمة بجانب اسم المتحول وليس النمط، كما في التعداد 31-56:

التعداد 31-56 **مثال سي ++ عن استخدام النجمة في التصريح عن مؤشر.**

```
EmployeeList *employees;
File *inputFile;
```

يمتلك هذا النهج ضعفاً هو أنه يوحي أن النجمة هي جزء من اسم المتحول، والتي ليست كذلك. يمكن أن يُستخدم المتحول إما مع أو بدون النجمة.

النهج الأفضل هو أن تصرح عن نمط للمؤشر وتستخدمه بدلاً. يظهر مثال عن ذلك في التعداد 31-57:

```
EmployeeListPointer employees;
FilePointer inputFile;
```

هذه المشكلة المحددة المعالجة بهذا النهج يمكن أن تُحل إما بفرض أن تَصرّح كل المؤشرات باستخدامك أنماط للمؤشرات، كما في التعداد 31-57، أو بفرض عدم كتابة أكثر من تصريح عن متحول على السطر الواحد. تأكد من أن تختار على الأقل واحد من هذين الحلين!

31. 6 رسم التعليقات

التعليقات المكتوبة بطريقة جيدة يمكن أن تُحسّن قابلية القراءة لبرنامج بشكل عظيم، والتعليقات المكتوبة برداءة يمكن أن تؤذيها فعلياً. يلعب تنسيق التعليقات دوراً عظيماً في كون التعليقات تساعد أو تعيق قابلية القراءة.[1]

[1] **إشارة مرجعية** لتفاصيل عن الجوانب الأخرى للتعليقات، الق نظرة على الفصل 32، "الشفرة الموثقة ذاتياً."

حاذِ التعليق وفق شفرته الموافقة المحاذاة المرئية هي مساعد قيّم في فهم بنيان البرنامج المنطقي، والتعليقات الجيدة لا تتعارض مع المحاذاة المرئية. مثلاً ما هو البنيان المنطقي للإجرائية الظاهرة في التعداد 31-58؟

التعداد 31-58 مثال فيجوال بيسك عن تعليقات حوذيت برداءة.




```
For transactionId = 1 To totalTransactions
' get transaction data
   GetTransactionType( transactionType )
   GetTransactionAmount( transactionAmount )

' process transaction based on transaction type
   If transactionType = Transaction_Sale Then
      AcceptCustomerSale( transactionAmount )

   Else
      If transactionType = Transaction_CustomerReturn Then

' either process return automatically or get manager approval, if required
         If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

' try to get manager approval and then accept or reject the return
' based on whether approval is granted
            GetMgrApproval( isTransactionApproved )
            If ( isTransactionApproved ) Then
               AcceptCustomerReturn( transactionAmount )
            Else
               RejectCustomerReturn( transactionAmount )
            End If
         Else

' manager approval not required, so accept return
            AcceptCustomerReturn( transactionAmount )
         End If
      End If
   End If
Next
```

في هذا المثال، لا تحصل على الكثير من الدلائل على البنيان المنطقي لأن التعليقات تجعل المحاذاة المرئية للشفرة غامضة بشكل كامل. قد تجد التصديق بأن أحداً ما يتخذ قراراً واعياً باستخدام أسلوب محاذة كهذا أمراً صعباً، لكني رأيته في برامج احترافية وأعرف كتاباً واحداً على الأقل ينصح به.

الشفرة الظاهرة في التعداد 31-49 هي تماماً مثل الموجودة في التعداد 31-48، باستثناء محاذاة التعليقات.

التعداد 31-59 مثال فيجوال بيسك عن تعليقات حوذيت بشكل كيّس

```
For transactionId = 1 To totalTransactions
   ' get transaction data
   GetTransactionType( transactionType )
   GetTransactionAmount( transactionAmount )

   ' process transaction based on transaction type
   If transactionType = Transaction_Sale Then
      AcceptCustomerSale( transactionAmount )

   Else
      If transactionType = Transaction_CustomerReturn Then

         ' either process return automatically or get manager approval, if required
         If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

            ' try to get manager approval and then accept or reject the return
            ' based on whether approval is granted
            GetMgrApproval( isTransactionApproved )
            If ( isTransactionApproved ) Then
               AcceptCustomerReturn( transactionAmount )
            Else
               RejectCustomerReturn( transactionAmount )
            End If
         Else

            ' manager approval not required, so accept return
            AcceptCustomerReturn( transactionAmount )
         End If
      End If
   End If
Next
```

في التعداد 31-59، البنيان المنطقي أكثر وضوحاً. وجدت دراسة واحدة عن فعالية كتابة التعليقات أن فائدة وجود تعليقات كان غير حاسم، وخمّن الكاتب أن ذلك بسبب أن التعليقات "تشوش المسح المرئي للبرنامج" (شنيدرمان 1980). من هذه الأمثلة، إنه لأمر واضح أن أسلوب كتابة التعليقات يؤثر بقوة على كون التعليقات مزعجة أو لا.

هيئ كل تعليق بسطر فارغ واحد على الأقل إن كان شخص ما يحاول أن يأخذ نظرة عامة عن برنامجك، فإن الطريقة الأكثر فعالية ليقوم بذلك هي أن يقرأ التعليقات بدون الشفرة. يساعد تهيئة التعليقات بسطور فارغة القارئ على مسح الشفرة. يُظهر التعداد 31-60 مثالاً:

التعداد 31-50 **مثال جافا عن تهيئة تعليق بسطر فارغ.**

```
// comment zero
CodeStatementZero;
CodeStatementOne;

// comment one
CodeStatementTwo;
CodeStatementThree;
```

يستخدم بعض الناس سطرين فارغين واحد قبل وآخر بعد التعليق. يستخدم السطران الفارغان مساحة عرض أكبر، لكن يعتقد بعض الناس أن الشفرة تبدو أفضل مما لو كانت عليه بواحد فقط. يُظهر التعداد 31-61 مثالاً:

التعداد 31-61 مثال جافا عن تهيئة تعليق بسطرين فارغين.

```
// comment zero

CodeStatementZero;
CodeStatementOne;

// comment one

CodeStatementTwo;
CodeStatementThree;
```

مالم تكن مساحة العرض الخاصة بك أغلى من المعتاد، فإن هذا محاكمة جمالية نقية وتستطيع أن تقوم بها تبعاً لذلك. في هذه المنطقة، كما في المناطق العديدة الأخرى، حقيقة وجود عرف أهم من تفاصيل العرف التفصيلية.

31. 7 رسم الإجرائيات

الإجرائيات مؤلفة من عبارات وبيانات وبنى تحكم وتعليقات-كل الأشياء التي نوقشت في الأجزاء الأخرى من هذا الفصل-مستقلة. يقدم هذا القسم توجيهات تنسيقية خاصة بالإجرائيات فقط.[1]

استخدم سطوراً فارغة لتفصل أجزاء الإجرائية استخدم سطوراً فارغة بين رأس الإجرائية وتصريحات بياناتها وثوابتها المسماة (إن وجدت)، وجسمها.

استخدم المحاذاة المعيارية لوسطاء الإجرائية خيارات تنسيق رأس الإجرائية هي تقريباً نفس الخيارات الموجودة في الكثير من المناطق الأخرى الخاضعة للتنسيق: لا تنسيق واع، أو تنسيق سطر النهاية، أو المحاذاة المعيارية. كما في معظم الحالات الأخرى، تقوم المحاذاة المعيارية بأفضل ما يمكن بالنسبة للدقة والتماسك وقابلية القراءة وقابلية التعديل. يُظهر التعداد 31-62 مثالين عن رأسي إجرائيتين بلا تنسيق واع:

التعداد 31- 62 مثالي سي ++ عن رأسي إجرائية بلا تنسيق واع.

[1] **إشارة مرجعية** لتفاصيل عن توثيق الإجرائيات، الق نظرة على القسم 32. 5. لتفاصيل عن عملية كتابة إجرائية، راجع القسم 9. 3، "بناء الإجرائيات باستخدام تربل ابي." لنقاش حول الفوارق بين الإجرائيات الجيدة والسيئة راجع القسم 7، "إجرائيات عالية الجودة."

```
bool ReadEmployeeData(int maxEmployees,EmployeeList *employees,
   EmployeeFile *inputFile,int *employeeCount,bool *isInputError)
...

void InsertionSort(SortArray data,int firstElement,int lastElement)
```

رأسا الإجرائيتين هذين هادفان إلى المنفعة على نحو محض. يستطيع الحاسوب أن يقرأهما كما يستطيع قراءة رؤوساً بأي تنسيق آخر، لكنهما يشكلان متاعب للبشر. بدون جهد واع لجعل الرأسين صعبي القراءة، كيف من الممكن أن يكونوا أسوأ؟

النهج الثاني في تنسيق رأس الإجرائية هو تنسيق سطر النهاية، والذي يعمل عادةً على ما يرام. يُظهر التعداد 31-63 نفس رأسي الإجرائيتين بعد أن أعيد تنسيقهما:

التعداد 31-63 مثال سي ++ عن رأسي إجرائيتين بتنسيق سطر النهاية العادي.

```
bool ReadEmployeeData( int                   maxEmployees,
                       EmployeeList          *employees,
                       EmployeeFile          *inputFile,
                       int                   *employeeCount,
                       bool                  *isInputError )
...
void InsertionSort( SortArray        data,
                    int              firstElement,
                    int              lastElement )
```

نهج سطر النهاية مهندم ومقبول من الناحية الجمالية[1]. مشكلته الرئيسة أنه يستهلك الكثير من العمل في الصيانة، والأساليب صعبة الصيانة لا تصان. افترض أن اسم التابع تغير من *ReadEmployeeData()* إلى *ReadNewEmployeeData()*. هذا سيرمي محاذاة السطر الأول بعيداً عن محاذاة السطور الأربعة الباقية. سيكون عليك أن تعيد تنسيق السطور الأربعة الأخرى في لائحة الوسطاء لتحاذيها مع الموقع الجديد ل *maxEmployees* الناتج عن اسم التابع الأطول. وقد تنفد من المساحة في الجهة اليمنى لأن العناصر هي في أقصى اليمين مسبقاً.

المثالين الظاهرين في التعداد 31-64، مُنسقيّن باستخدام المحاذاة المعيارية، وهما تماماً بدرجة القبول الجمالي للتنسيق السابق وهما يستهلكان عملاً أقل للصيانة.

[1] **إشارة مرجعية** لمزيد من التفاصيل عن استخدام وسطاء الإجرائية، راجع القسم 7. 5، "كيف تستخدم وسطاء الإجرائية."

التعداد 31 64 مثال سي ++ عن رأسي إجرائيتين مع محاذاة معيارية قابلة للقراءة والصيانة.

```
bool ReadEmployeeData(
   int maxEmployees,
   EmployeeList *employees,
   EmployeeFile *inputFile,
   int *employeeCount,
   bool *isInputError
)
...

void InsertionSort(
   SortArray data,
   int firstElement,
   int lastElement
)
```

يصمد هذا الأسلوب بشكل أفضل تحت التعديلات. إن تغير اسم الإجرائية، فلن يكون للتغير أثر على أي من المتحولات. إن أُضيفت متحولات أو حذفت، فيجب تعديل سطر واحد فقط-زائد أو ناقص فاصلة. الدلائل المرئية مشابهة لتلك التي في تخطيط محاذاة حلقة أو عبارة *if*. لا يجب على عينيك أن تمسح أجزاء مختلفة من الصفحة من أجل كل إجرائية مفردة كي تجد معلومات مفيدة؛ إنها تعرف أين تكون المعلومة في كل مرة.

هذا النموذج يُترجم إلى **فيجوال بيسك** بطريقة مباشرة، مع أنه يتطلب استخدام محارف إكمال السطر، كما يظهر في التعداد 31-65:

التعداد 31-65 مثال فيجوال بيسك عن رأس رؤوس إجرائية بمحاذاة معيارية قابلة للصيانة والقراءة.

```
Public Sub ReadEmployeeData ( _
   ByVal maxEmployees As Integer, _
   ByRef employees As EmployeeList, _
   ByRef inputFile As EmployeeFile, _
   ByRef employeeCount As Integer, _
   ByRef isInputError As Boolean _
)
```

ها هنا محرف "_" مُستخدم كمحرف إكمال السطر.

31. 8 رسم الصفوف

يعرض هذا القسم توجيهات لرسم الشفرة في الصفوف. يصف القسم الفرعي الأول كيف ترسم واجهة صف. والثاني كيف ترسم تحقيق صف. ويناقش الأخير رسم الملفات والبرامج.

رسم واجهات الصفوف[1]

في تنسيق واجهات الصفوف، العرف هو أن تأتي بأعضاء الصف بالترتيب التالي:

[1] **إشارة مرجعية** لتفاصيل عن توثيق الصفوف، الق نظرة على " توثيق الصفوف والملفات والبرامج" في القسم 32. 5. لنقاش عن الفوارق بين الصفوف الجيدة والسيئة، راجع الفصل 6، "صفوف ناجحة".

1. التعليقات على الرأس التي تصف الصف وتقدم أية ملاحظة عن الاستخدام الكلي للصف
2. البواني والهوادم
3. الإجرائيات العامة
4. الإجرائيات المحمية
5. الإجرائيات والبيانات الأعضاء الخاصة

رسم تحقيقات الصفوف

تُنسّق تحقيقات الصفوف بشكل عام في هذا الترتيب:

1. التعليقات على الرأس التي تصف محتوى الملف الذي فيه الصف
2. بيانات الصف
3. الإجرائيات العامة
4. الإجرائيات المحمية
5. الإجرائيات الخاصة

إن كان لديك أكثر من صف واحد في الملف، عرّف كل صف بوضوح ينبغي أن تُجمَّع الإجرائيات المترابطة مع بعضها في صفوف. وينبغي أن يكون القارئ الذي يفحص شفرتك قادراً على تمييز الصفوف عن بعضها. عرّف كل صف بشكل واضح باستخدام عدة سطور فارغة بينه وبين الصف الذي يليه. الصف مثل فصل في كتاب. في كتاب، تبدأ كل فصل بصفحة جديدة وتستخدم طباعة كبيرة لعنوان الفصل. أكّد على بداية كل صف بشكل مشابه. يظهر مثال عن فصل الصفوف في التعداد 31-66:

التعداد 31-66 مثال سي ++ عن تنسيق العزل بين الصفوف

```
// create a string identical to sourceString except that the
// blanks are replaced with underscores.
void EditString::ConvertBlanks(
   char *sourceString,
   char *targetString
   ) {
   Assert( strlen( sourceString ) <= MAX_STRING_LENGTH );
   Assert( sourceString != NULL );
   Assert( targetString != NULL );
   int charIndex = 0;
   do {
      if ( sourceString[ charIndex ] == " " ) {
         targetString[ charIndex ] = '_';
      }
      else {
         targetString[ charIndex ] = sourceString[ charIndex ];
      }
      charIndex++;
   } while sourceString[ charIndex ] != '\0';
}

//----------------------------------------------------------------------
// MATHEMATICAL FUNCTIONS
//
// This class contains the program's mathematical functions.
//----------------------------------------------------------------------

// find the arithmetic maximum of arg1 and arg2
int Math::Max( int arg1, int arg2 ) {
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}

// find the arithmetic minimum of arg1 and arg2
int Math::Min( int arg1, int arg2 ) {
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

هذه هي آخر إجرائية في صف

بداية الصف الجديد مُعلّمة بعدة سطور فارغة واسم الصف

هذه هي أول إجرائية في الصف الجديد

هذه الإجرائية مفصولة عن الإجرائية السابقة بسطور فارغة فقط

تجنب التأكيد الزائد على التعليقات ضمن الصفوف، إن علّمت كل إجرائية وتعليق بسطر من النجوم بدلاً من السطور الفارغة، ستقضي وقتاً صعباً في اختراع أداة حيلة تؤكد بشكل فعال بداية سطر جديد. يظهر مثال في التعداد 31-67:

التعداد 31-67 مثال سي ++ عن تهيئة زائدة لصف.

```
// ******************************************************************
// ******************************************************************
// MATHEMATICAL FUNCTIONS
//
// This class contains the program's mathematical functions.
// ******************************************************************
// ******************************************************************

// ******************************************************************
// find the arithmetic maximum of arg1 and arg2
// ******************************************************************
int Math::Max( int arg1, int arg2 ) {
// ******************************************************************
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}

// ******************************************************************
// find the arithmetic minimum of arg1 and arg2
// ******************************************************************
int Math::Min( int arg1, int arg2 ) {
// ******************************************************************
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

في هذا المثال، الكثير جداً من الأشياء وسمت بالنجوم لدرجة أن لا شيء فعلياً تم تأكيده. أصبح البرنامج غابة كثيفة من النجوم. مع أنها محاكمة جمالية أكثر من كونها محاكمة تقنية، في التنسيق، الزائد أخ للناقص.

إن توجب حتماً عليك أن تفصل أجزاء من البرنامج بسطور طويلة من محارف خاصة، طّور هرمية من المحارف (من الأثقل إلى الأخف) بدلاً من الاعتماد الحصري على النجوم. مثلاً، استخدم نجوماً لتقسيم الصفوف، وشحطات "-" لتقسيم الإجرائيات، وسطور فارغة للتعليقات المهمة. انتهِ عن وضع سطرين من النجوم أو الشحطات. يَظهر مثال في التعداد 31-68:

التعداد 31-68 **مثال سي ++ عن تهيئة جيدة مع تحفّظ.**

```
// ******************************************************************
// MATHEMATICAL FUNCTIONS
//
// This class contains the program's mathematical functions.
// ******************************************************************

// ------------------------------------------------------------------
// find the arithmetic maximum of arg1 and arg2
// ------------------------------------------------------------------
int Math::Max( int arg1, int arg2 ) {
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}

// ------------------------------------------------------------------
// find the arithmetic minimum of arg1 and arg2
// ------------------------------------------------------------------
int Math::Min( int arg1, int arg2 ) {
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

خفة هذا السطر بالمقارنة مع سطر النجوم تدعم من الناحية المرئية حقيقة أن الإجرائية تابعة للصف

هذه النصيحة عن كيفية تعريف عدة صفوف ضمن ملف مفرد تُطبَّق فقط عندما تحدد لغتك عدد الملفات التي يمكن أن تستخدمها في برنامج. إن كنت تستخدم **سي ++ أو جافا أو فيجوال بيسك** أو أي لغة أخرى تدعم تعدد الملفات المصدرية، ضع صفاً واحداً فقط في كل ملف مالم يكن عندك سبب قاهر لتقوم بغير هذا (مثل تضمين عدة صفوف صغيرة تُكوِّن نموذجاً واحداً). ضمن صف مفرد، على كل، قد لا يزال لديك مجموعات جزئية من الإجرائيات، وبإمكانك أن تُجمّعها باستخدام تقنيات مثل التقنيات الظاهرة هنا.

رسم الملفات والبرامج[1]

خلف تقنيات التنسيق للصفوف توجد قضية تنسيق أكبر: كيف تُنظِّم الصفوف والإجرائيات ضمن ملف، وكيف تقرر أي الصفوف تضع معاً في ملف في المقام الأول؟

[1] **إشارة مرجعية** من أجل تفاصيل التوثيق، الق نظرة على "التعليق على الصفوف والملفات والبرامج" في القسم 32 .5.

ضع صفاً واحداً في ملف واحد ليس الملف جيبة تحمل بعض الشفرة. إن كانت لغتك تسمح، ينبغي أن يحمل الملف مجموعة من الإجرائيات التي تدعم غرضاً واحداً وفقط واحداً. يدعم الملف فكرة أن مجموعة من الإجرائيات توجد في نفس الصف.

كل الإجرائيات ضمن الملف تُكوِّن الصف[1]. قد يكون الصف واحداً يتعرف عليه البرنامج فعلياً كذلك، أو قد يكون مجرد كينونة منطقية أنشأتها كجزء من تصميمك.

الصفوف هي مفاهيم معنوية متعلقة باللغة. بينما الملفات هي مفاهيم فيزيائية متعلقة بنظام التشغيل. التوافق بين الصفوف والملفات هو صدفة وهو يستمر بالضعف مع مرور الوقت طالما أن المزيد من البيئات تدعم وضع الشفرة ضمن قاعدة بيانات أو بشكل آخر تجعل العلاقة بين الإجرائيات والصفوف والملفات غامضة.

أعط الملف اسماً ذو صلة باسم الصف تمتلك معظم المشاريع توافق واحد لواحد بين أسماء الصفوف وأسماء الملفات. صف اسمه *CustomerAccount* يوافقه ملف اسمه *CustomerAccount.cpp* وملف اسمه *CustomerAccount.h*، مثلاً.

افصل بين الإجرائيات ضمن الملف بشكل واضح اعزل كل إجرائية عن الإجرائيات الأخرى بسطرين فارغين على الأقل. السطور الفارغة بفعالية سطور كبيرة من النجوم أو الشحطات، وهي أسهل كثيراً كتابة وصيانةً. استخدم اثنين أو ثلاثة لتصنع فرقاً مرئياً بين السطور الفارغة التي هي جزء من الإجرائية والسطور الفارغة التي تفصل الإجرائيات. كما في المثال الظاهر في التعداد 31-69:

[1] **إشارة مرجعية** لتفاصيل عن الفوارق بين الصفوف والإجرائيات وكيفية صنع مجموعة من الإجرائيات ضمن صف، راجع الفصل 6، "صفوف ناجحة."

التعداد 31-69 **مثال فيجوال بيسك عن استخدام سطور فارغة بين الإجرائيات.**

```
'find the arithmetic maximum of arg1 and arg2
Function Max( arg1 As Integer, arg2 As Integer ) As Integer
   If ( arg1 > arg2 ) Then
      Max = arg1
   Else
      Max = arg2
   End If
End Function



'find the arithmetic minimum of arg1 and arg2
Function Min( arg1 As Integer, arg2 As Integer ) As Integer
   If ( arg1 < arg2 ) Then
      Min = arg1
   Else
      Min = arg2
   End If
end Function
```


السطور الفارغة أسهل كتابةً من أي نوع آخر من الفواصل وتبدو على الأقل بجودتها. استُخدمت ثلاثة سطور فارغة في هذا المثال لذلك أصبح العزل بين الإجرائيتين لافتاً للانتباه أكثر من السطور الفارغة ضمن كل إجرائية.

افصل الإجرائيات أبجدياً وضع الإجرائيات بترتيب أبجدي هو بديل لتجميع الإجرائيات المترابطة في ملف. إن لم تستطع أن تُجزّء برنامجاً إلى صفوف أو إن لم يسمح لك المحرر الذي تستخدمه بإيجاد التوابع بسهولة، يستطيع النهج الأبجدي توفير وقت البحث.

في سي++، رتب الملفات المصدرية بحذر ها هنا ترتيب نمطي لمحتويات الملفات المصدرية في سي++:

1. تعليقات توصيف الملف
2. ملفات include #
3. تعريفات الثوابت التي تُطبَّق على أكثر من صف واحد (إن احتوى الملف على أكثر من صف واحد)
4. التعداديات (enums) التي تُطبَّق على أكثر من صف واحد (إن احتوى الملف على أكثر من صف واحد)
5. تعريفات توابع الماكرو
6. تعريفات الأنماط التي تُطبَّق على أكثر من صف واحد (إن احتوى الملف على أكثر من صف واحد)
7. المتحولات الشاملة والتوابع المستوردة
8. المتحولات الشاملة والتوابع المصدرة

9. المتحولات والتوابع الخاصة بالملف
10. الصفوف، متضمنة تعاريف الثوابت، والتعداديات، وتعاريف الأنماط ضمن كل صف

لائحة اختبار: التنسيق[1]

عام

- هل قمت بالتنسيق بشكل رئيس للإضاءة على البنيان المنطقي للشفرة؟
- هل من الممكن استخدام مخطط التنسيق بتماسك؟
- هل يعطي مخطط التنسيق شفرة سهلة الصيانة؟
- هل يُحسّن مخطط التنسيق قابلية القراءة؟

بنى التحكم

- هل تتجنب الشفرة أزواج *begin-end* أو {} المسبوقة بمسافات بادئة بشكل مضاعف؟
- هل الكتل المتتالية مفصولة عن بعضها البعض بسطور فارغة؟
- هل نُسّقت التعابير المعقدة من أجل قابلية القراءة؟
- هل نُسّقت "كتل العبارات المفردة" بشكل متماسك؟
- هل نُسّقت عبارات *case* بطريقة متماسكة مع تنسيق بنى التحكم الأخرى؟
- هل نُسّقت *gotos* بطريقة تجعل استخدامها واضحاً؟

العبارات المفردة

هل استخدمت المسافات الفارغة لتجعل التعابير المنطقية، وإشارات المصفوفات، ووسطاء الإجرائيات مقروءة؟

هل تُنهي العبارات غير الكاملة السطر بطريقة تبدو فيها غير صحيحة بشكل واضح؟

هل حوذيت سطور الإكمال بالمقدار المعياري للمحاذاة؟

هل يحتوي كل سطر عبارة واحدة على الأكثر؟

هل كُتبت كل عبارة بدون آثار جانبية؟

هل يوجد تصريح واحد عن البيانات على الأكثر على السطر الواحد؟

التعليقات

[1] cc2e.com/3194

هل سُبقت التعليقات بنفس عدد الفراغات الموجودة أمام الشفرة التي تُعلّق عليها؟

هل أسلوب التعليق سهل الصيانة؟

الإجرائيات

- هل نُسِّقت وسطاء كل إجرائية بحيث يكون كل وسيط سهل القراءة والتعديل، والتعليق عليه سهل أيضاً؟
- هل استُخدمت السطور الفارغة لفصل أجزاء الإجرائية؟

الصفوف والملفات والبرامج

- هل توجد علاقة واحد لواحد بين الصفوف والملفات من أجل معظم الصفوف والملفات؟
- إن احتوى ملف عدة صفوف فعلياً، هل كل الإجرائيات في كل صف جُمِّعت مع بعضها وهل كل صف مُحدَّد بشكل واضح؟
- هل عزلت الإجرائيات ضمن الملف بسطور فارغة بشكل واضح؟
- كبديل لمبدأ التنظيم الأقوى، هل كل الإجرائيات في ترتيب أبجدي؟

مصادر إضافية[1]

تقول معظم الكتب البرمجية فقط بضعة كلمات عن التنسيق والأسلوب، لكن النقاشات العميقة عن أسلوب البرمجة نادرة؛ والنقاشات عن التنسيق هي أندر وأندر. الكتب التالية تتحدث عن التنسيق وأسلوب البرمجة:

"ممارسة القراءة البرمجية"

Kernighan, Brian W. and Rob Pike. *The Practice of Programming* Reading, MA: AddisonWesley, 1999. الفصل 1 من هذا الكتاب يناقش أسلوب البرمجة مركزاً على **سي وسي** ++.

"عناصر أسلوب الجافا"

Vermeulen, Allan, et al. *The Elements of Java Style*. Cambridge University Press, 2000.

"عناصر أسلوب السي++"

Misfeldt, Trevor, Greg Bumgardner, and Andrew Gray. *The Elements of C++ Style*. Cambridge University Press, 2004

-"عناصر أسلوب البرمجة" الإصدار الثاني

[1] cc2e.com/3101

Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*, 2d ed. New York, NY: McGraw-Hill, 1978. هذا كتاب تقليدي في أسلوب البرمجة-الأول في الكتب من نوع أسلوب البرمجة.

من أجل نُهج قابلية القراءة المختلفة بصورة ملحوظة، الق نظرة على الكتاب التالي:

"البرمجة المثقفة (الأدبية)"

Knuth, Donald E. Literate Programming. Cambridge University Press, 2001. هذا عبارة عن تجميعة من المقالات التي تصف نَهج "البرمجة المثقفة" لدمج لغة البرمجة ولغة التوثيق. كان نَث يكتب عن مناقب البرمجة المثقفة لحوالي 20 سنة، لكن رغم مطالبته القوية بلقب **المبرمج الأفضل على الكوكب**، لم تنتشر البرمجة المثقفة. اقرأ بعضاً من شفرته لتكوّن استنتاجاتك الخاصة حول السبب.

نقاط مفتاحية

- الأولوية الأولى للتنسيق المرئي هي الإضاءة على التنظيم المنطقي للشفرة. المعيار المستخدم لتخمين إن كانت هذه الأولوية مُنجَزة يتضمن الدقة والتماسك وقابلية القراءة وقابلية الصيانة.
- أن تبدو جيداً هو أمر ثانوي بالنسبة للمعايير الأخرى-هو الثاني بعد مسافة كبيرة. إن طيعت المعايير الأخرى وكانت الشفرة في الطبقة الأدنى شفرة جيدة، على كل، سيبدو التنسيق كيّساً.
- تمتلك **فيجوال بيسك** الكتل النقية والتطبيق العرفي في **جافا** هو ا ستخدام أ سلوب الكتل النقية، لذا بإمكانك استخدام تنسيق الكتل النقية إن كنت تبرمج بهاتين اللغتين. في **سي** ++، إما محاكاة الكتل النقية أو حدود الكتلة *begin-end* ستعمل بشكل جيد.
- هيكلة الشفرة هامة للشفرة نفسها. العرف المحدد الذي تتبعه أقل أهمية من حقيقة اتباعك لنفس العرف بثبات. يمكن لعرف التنسيق غير المُتَّبَع بثبات أن يؤذي قابلية القراءة.
- العديد من جوانب التنسيق هي قضايا مذهبية. حاول أن تفصل التفضيلات الموضوعية عن التفضيلات الشخصية. استخدم معياراً واضحاً لتساعد في تأريض نقاشاتك حول خيارات الأسلوب.

الفصل 32

الشفرة الموثَّقة ذاتياً

المحتويات[1]

مواضيع ذات صلة

- التنسيق: الفصل 31
- عملية البرمجة بالشفرة الزائفة: الفصل 9
- صفوف ناجحة: الفصل 6
- إجرائيات عالية الجودة: الفصل 7
- البرمجة كتواصل: القسمين 33. 5 و34. 3

يستمتع معظم المبرمجين بكتابة التوثيق عندما تكون معايير التوثيق منطقية[2]. مثل التنسيق، التوثيق الجيد هو علامة على المفخرة الاحترافية، يضعها المبرمج في برنامجه. يمكن أن يأخذ توثيق الشفرة عدة أشكال، و، يهتم هذا الفصل بالمنطقة الخاصة من التوثيق المعروفة ب "التعليقات،" بعد وصف الطريق الطويل للمنظر الطبيعي للتوثيق.

---

[1] cc2e.com/3245

[2] برمج وكأن أي شخص يقوم بصيانة برنامجك هو شخص عنيف ومضطرب عقلياً ويعلم أين تعيش –مجهول

32. 1 التوثيقات الخارجية[1]

تتألف التوثيقات على المشاريع البرمجية من معلومات من داخل تعداد الشفرة المصدرية وخارجه-عادةً في صيغة وثائق منفصلة أو مجلدات تطوير الوحدة. في المشاريع الضخمة والرسمية معظم التوثيقات تكون خارج الشفرة المصدرية (جونز 1998). تميل توثيقات البناء الخارجية لتكون في مستويات عليا مشابهة للشفرة، وفي مستويات دنيا مشابهة للتوثيقات من تعريف المشكلة، والمتطلبات، ونشاطات الهيكلة.

مجلدات تطوير الوحدة[2] مجلد تطوير الوحدة (UDF)، أو مجلد تطوير البرمجية (SDF)، هو وثيقة غير رسمية تحتوي ملاحظات استخدمها المطور خلال البناء. "الوحدة" معرفة بفضاضة، عادةً تعني الصف، مع أنها يمكن أن تعني الحزمة أو المكون. الغرض الرئيس لمجلد تطوير الوحدة هو أن يؤمن أثرا لقرارات التصميم التي لا تُوثَّق في أي مكان آخر. تمتلك العديد من المشاريع معاييراً تحدد المحتوى الأصغري لمجلد تطوير الوحدة، كنسخ للمتطلبات ذات الصلة، وأجزاء من تصميم المستوى الأعلى الذي تحققه الوحدة، ونسخة من معايير التطوير، وتعداد حالي للشفرة، وملاحظات التصميم من مطور الوحدة. أحياناً، يتطلب الزبون من مطور البرمجية أن يسلم مجلدات تطوير الوحدة الخاصة بالمشروع، وهي غالباً للاستخدام الداخلي فقط.

وثيقة التصميم التفصيلي وثيقة التصميم التفصيلي هي وثيقة التصميم في المستوى المنخفض. إنها تصف قرارات التصميم في مستوى الصف أو مستوى الإجرائية، والبدائل التي كانت مُعتبَرة، وأسباب اختيار النُهج التي اختيرت. أحياناً تُضمَّن هذه المعلومات في وثيقة رسمية. في هكذا حالات، عادةً ما يُعتبر التصميم التفصيلي منفصلاً عن البناء. وأحياناً تتكون بشكل رئيس من ملاحظات المطورين المُجمَّعة في ملف تطوير الوحدة. وأحيانا-غالباً-توجد فقط في الشفرة نفسها.

32. 2 أسلوب البرمجة كتوثيق

في مقابل التوثيق الخارجي، يوجد التوثيق الداخلي ضمن تعداد البرنامج نفسه. إنه النوع الأكثر تفصيلاً من التوثيق، في مستوى العبارات المصدرية. وبسبب أنها الأقرب صلة بالشفرة، التوثيق الداخلي هو أيضاً نوع التنسيق ذي الاحتمال الأكبر بأن يبقى صحيحاً بينما تتعدل الشفرة.

المساهم الرئيس في التوثيق على مستوى الشفرة ليس التعليقات، بل أسلوب البرمجة الجيد. يتضمن الأسلوب: الهيكلة الجيدة للبرنامج، وأسماء المتحولات الجيدة، وأسماء الإجرائيات الجيدة، واستخدام الثوابت المسماة بدلاً من الحرفيّات، والتنسيق الواضح، وتخفيض تعقيد تدفق التحكم وهيكلة البيانات.

[1] **إشارة مرجعية** للمزيد عن التوثيقات الخارجية الق نظرة على القسم 32. 6. "معايير آي تربل إي."

[2] **إشارة مرجعية** من أجل توصيف تفصيلي، الق نظرة على "مجلد تطوير الوحدة (UDF): أداة إدارة فعالة لتطوير البرمجيات" (إنغريسيا 1976) أو "مجلد تطوير الوحدة (UDF)؛ وجهة نظر مستخلصة من عشر سنوات" (إنغريسيا 1987).

هاهنا قطعة شفرة ذات أسلوب رديء:

مثال **جافا** عن توثيق رديء، ينتج من أسلوب برمجة سيء

```java
for ( i = 2; i <= num; i++ ) {
meetsCriteria[ i ] = true;
}
for ( i = 2; i <= num / 2; i++ ) {
j = i + i;
while ( j <= num ) {
meetsCriteria[ j ] = false;
j = j + i;
}
}
for ( i = 2; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets criteria." );
}
}
```

ماذا تعتقد أن هذه الإجرائية تفعل؟ إنها مبهمة بلا ضرورة لتكون كذلك. إنها مُنسَّقة برداءة ليس لأنها تفتقد التعليقات، بل لأنها تفتقد أسلوب البرمجة الجيد. أسماء المتحولات غير ذات دلالة، والتنسيق مزدحم. ها هنا نفس الشفرة مُحسَّنةً-تحسين أسلوب البرمجة فقط، يجعل معناها أوضح بكثير:

مثال **جافا** عن تنسيق بدون تعليقات (بأسلوب جيد)

```java
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
   int factorableNumber = factor + factor;
   while ( factorableNumber <= num ) {
      isPrime[ factorableNumber ] = false;
      factorableNumber = factorableNumber + factor;
   }
}

for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   if ( isPrime[ primeCandidate ] ) {
      System.out.println( primeCandidate + " is prime." );
   }
}
```

ليست مثل القطعة الأولى من الشفرة[1]، تجعلك هذه القطعة تعرف من النظرة الأولى أنها ذات صلة بالأعداد الأولية. تكشف النظرة الثانية أنها تجد الأعداد الأولية بين 1 و *Num*. بالنسبة لقطعة الشفرة الأولى، إنها تأخذ أكثر من مجرد نظرتين لتكتشف اين تنتهي الحلقات.

[1] **إشارة مرجعية** في هذه الشفرة، أضيف المتحول *factorableNumber* فقط من أجل توضيح العملية. لتفاصيل حول إضافة متحولات لتوضيح العمليات، راجع "جعل التعابير المعقدة بسيطة" في القسم 19. 1.

الفرق بين مقطعي الشفرة ليس له صلة بالتعليقات-ولا أي مقطع فيه أي تعليق. الثاني أكثر قابلية للقراءة بكثير، على كل، ويقارب الكاس المقدسة للوضوح: الشفرة الموثقة ذاتياً. تعتمد هكذا شفرة على أسلوب البرمجة الجيد لتَحمِل الجزء الأعظم من حمل التوثيق. في شفرة مكتوبة جيداً، التعليقات هي الكريمة على قالب كعكة قابلية القراءة.

لائحة اختبار: الشفرة الموثقة ذاتياً[1]

الصفوف

- هل تقدم واجهة كل صف تجريداً متماسكاً؟
- هل كل صف مسمى بشكل جيد، وهل يصف اسمه غايته المركزية؟
- هل توضح واجهة كل صف الكيفية التي ينبغي بها أن تستخدم الصف؟
- هل واجهة كل صف مجردة إلى حد كاف يُمكّنك ألا تضطر إلى التفكير بكيفية تحقيق خدماتها؟ هل تستطيع أن تُعامل الصف كصندوق أسود؟

الإجرائيات

- هل تصف أسماء كل الإجرائيات بالضبط ما تقوم به كل إجرائية؟
- هل تقوم كل إجرائية بمهمة واحدة معرفة جيداً؟
- هل كل الأجزاء، في كل الإجرائيات، التي ستجلب النفع بوضعها في إجرائيات خاصة، وُضعت في إجرائيات خاصة؟
- هل واجهات كل الإجرائيات بيّنة وواضحة؟

أسماء البيانات

- هل أسماء الأنماط معبِّرة إلى حد كافٍ لتساعد في توثيق التصريحات عن البيانات؟
- هل المتحولات مُسماة بشكل جيد؟
- هل استخدِمت المتحولات فقط للغاية التي سُمّيت من أجلها؟
- هل أعطيت عدادات الحلقات أسماء ذات دلالة أكثر من *i, j, k*؟
- هل استخدِمت الأنماط التعدادية المعرفة بشكل جيد بدلاً من الرايات المؤقتة أو المتحولات المنطقية؟
- هل استخدِمت الثوابت المسماة بدلاً من الأرقام السحرية أو السلاسل النصية السحرية؟
- هل تفرق أعراف التسمية بين أسماء الأنماط، والأنماط التعدادية، والثوابت المسماة، والمتحولات المحلية، ومتحولات الصف، والمتحولات الشاملة؟

[1] cc2e.com/3252

تنظيم البيانات

- هل استخدِمت متحولات إضافية عند الضرورة من أجل التوضيح؟
- هل ورودات "ظهورات" كل متحول قريبة من بعضها؟
- هل أنماط البيانات بسيطة بحيث تُخفّض التعقيد؟
- هل يتم النفاذ إلى البيانات المعقدة عبر إجرائيات نفاذ مجردة (أنماط بيانات مجردة)؟

التحكم

- هل المسار الرئيس عبر الشفرة واضح؟
- هل العبارات المترابطة مُجمَّعة مع بعضها؟
- هل المجموعات المستقلة نسبياً من العبارات حُزِّمت إلى إجرائيات خاصة؟
- هل الحالة الطبيعية تتبع *if* وليس *else*؟
- هل بنى التحكم بسيطة بحيث تُخفِّض التعقيد؟
- هل تقوم كل حلقة بوظيفة واحدة وفقط واحدة، كما تفعل الإجرائية المعرفة بشكل جيد؟
- هل خُفِّض التعشيش؟
- هل بُسِّطت التعابير المنطقية باستخدام متحولات منطقية إضافية، أو توابع منطقية، أو جداول القرارات؟

التنسيق

هل يُظهر تنسيق البرنامج بنيانه المنطقي؟

التصميم

هل الشفرة مستقيمة، وتتجنب التذاكي؟

هل خُبِّئت تفاصيل التحقيق قدر الإمكان؟

هل كتب البرنامج بمفردات مجال المشكلة قدر الإمكان وليس بمفردات علم الحاسوب أو بنى لغة البرمجة؟

32.3 أن تُعلَق أو ألا تُعلَق

أن تُكتب التعليقات برداءة أسهل من أن تُكتب بجودة، ويمكن أن تكون كتابة التعليقات مؤذية أكثر مما تكون مفيدة. النقاشات الحامية حول فضائل كتابة التعليقات عادة تبدو كنقاشات فلسفية حول الفضائل الأخلاقية، ما يجعلني أعتقد أنه لو كان سقراط مبرمج للحاسوب، لكان سيجري بينه وبين تلاميذه النقاش التالي:

التعليق

الشخصيات:

تراسيماكوس شخص قليل الخبرة على المذهب الصفائي النظري يصدّق كل شيء يقرؤه

كاليكليس محارب قديم من المدرسة القديمة تحمّل أعباء الحرب-مبرمج "حقيقي"

غلاكون مبرمج شاب واثق لامع يتميز بكتابة البرامج الضخمة واستخدام القوى العمياء يجد حلول لكل شيء

إزمين مبرمجة كبيرة تعبت من الوعود الكبيرة، وفقط تبحث عن قلة من التطبيقات التي تعمل

سقراط المبرمج القديم الحكيم

موضوعة المسرحية

نهاية اجتماع الفريق اليومي السريع

"ألدى أحدكم أي قضية أخرى قبل أن نعود إلى العمل؟" سأل سقراط.

"أريد أن أقترح معيار كتابة تعليقات لمشروعنا،" قال تراسيماكوس. "بعض من مبرمجينا نادراً ما يعلقون على شفرتهم، وكلنا يعلم أن الشفرة بدون تعليقات غير قابلة للقراءة."

"لا بد أن تكون تخرجت من الكلية للتو وليس كما توقعت،" أجاب كاليكليس. "التعليقات هي ترياق أكاديمي، لكن أي شخص قام بأي برمجة حقيقية يعلم أن التعليقات تجعل الشفرة أصعب قراءةً، لا أسهل. الإنكليزية أقل دقة من **جافا** أو **فيجوال بيسك** وتذهب باتجاه الكثير من الحشو الزائد عن الحد. عبارات لغة البرمجة قصيرة وهادفة. إن لم تستطع جعل الشفرة واضحة، كيف تستطيع أن تجعل التعليقات واضحة؟ بالإضافة، تنتهي صلاحية التعليقات بينما تتغير الشفرة، إن صدّقت تعليقاً منتهي الصلاحية، تكون غرقت."

"أنا أتفق مع هذا،" تدخل غلاكون. "الشفرة المعلق عليها بكثافة هي أصعب قراءة لأنها تعني أن تقرأ أكثر. يجب عليّ مسبقاً أن أقرأ الشفرة؛ فلم ينبغي أن تكون قراءة الكثير من التعليقات واجبة عليّ، أيضاً؟"

"انتظر دقيقة." قالت إزمين، واضعة فنجان قهوتها لتتدخل بما يساوي دراكماسين (عملتين يونانيتين). "أعلم أن كتابة التعليقات يمكن أن تتم بطريقة خاطئة، لكن التعليقات الجيدة تساوي بوزنها ذهباً. قد وجبت عليّ صيانة شفرة تحتوي تعليقات وشفرة لا تحتوي، وفضّلت صيانة الشفرة ذات التعليقات. لا أعتقد أنه ينبغي أن يكون لدينا معياراً يقول استخدم تعليقاً واحداً لكل س سطراً من الشفرة، لكن ينبغي أن نُشجّع أي شخص على كتابة التعليقات."

"إن كانت التعليقات مضيعة للوقت، لمَ يستخدمها أحد الناس، كاليكليس؟" سأل سقراط.

"إما لأنها مطلب حتمي أو لأنه قرأ في مكان ما أنها مفيدة. لا أحد يفكر بها يمكن أن يقرر أنها مفيدة أبداً."

"اعتقدت إزمين أنها مفيدة. كانت هنا مذ ثلاث سنين، وهي تقوم بصيانة شفرتك التي لا تحوي تعليقات وشفرة أخرى تحوي، وفضّلت الشفرة ذات التعليقات. ماذا تعتقد من هذا؟"

"التعليقات عديمة النفع لأنها تكرر الشفرة بإسهاب أكثر فقط—"



"انتظر هنا تماماً،" قاطع تراسيماكوس. "التعليقات الجيدة لا تكرر الشفرة ولا تشرحها. إنها توضح الغاية منها. ينبغي أن تشرح التعليقات، في مستوى من التجريد أعلى من مستوى الشفرة، ما تحاول أن تقوم به."

"صحيح،" قالت إزمين. "لقد ألقيت نظرة على التعليقات لأجد القسم الذي يقوم بما عليّ أن أغيّر أو أصلح. إنك محق في أن التعليقات التي تكرر الشفرة لا تساعد على الإطلاق لأن الشفرة تقول كل شيء مسبقاً. عندما أقرأ التعليقات، أريد أن تكون مثل قراءة عناوين في كتاب أو فهرس المحتويات. تساعدني التعليقات في إيجاد القسم الصحيح، وبعدها أبدأ قراءة الشفرة. أن أقرأ جملة واحدة بالإنكليزية أسرع بكثير من أن أحلل 20 سطر من الشفرة المكتوبة بلغة برمجة." صبت إزمين لنفسها كوباً آخر من القهوة.

"اعتقد أن الناس الذين يرفضون أن يكتبوا تعليقات (1) يظنون أن شفرتهم أوضح مما يمكن أن تكون، أو (2) يظنون أن المبرمجين الآخرين مهتمين بشفرتهم أكثر بكثير مما هم عليه، أو (3) يظنون أن المبرمجين الآخرين أذكى مما هم عليه، أو (4) هم كسولون، أو (5) يخافون أن يكتشف أحد ما كيف تعمل شفرتهم.

"ستكون مراجعات الشفرة مساعداً كبيراً هنا، سقراط،" تابعت إزمين. "إن ادّعى أحدً أنها لا تحتاج إلى كتابة تعليقات وقُصف بالأسئلة خلال مراجعة-عندما يبدأ العديد من النظراء بالقول، "ما الجحيم الذي تحاول أن تقوم به في هذه القطعة من الشفرة؟"-بعدها سيبدؤون بإدخال التعليقات. إن لم يفعلوا ذلك بمفردهم، على الأقل سيكون لدى مديرهم الذخيرة ليجعلهم يفعلوا ذلك.

"أنا لا أتهمك بأنك كسول أو تخاف أن يكتشف الناس شفرتك، كاليكليس. لقد عملت على شفرتك وأنت واحد من أفضل المبرمجين في الشركة. لكن كن ذا قلب، امم؟ سيكون العمل على شفرتك أسهل عليّ إن استخدمت التعليقات."

"لكنها مضيعة للموارد،" ردّ كاليكليس. "ينبغي أن تكون شفرة المبرمج الجيد موثقة ذاتياً، ينبغي أن يكون كل شيء تحتاجينه في الشفرة."

"مستحيل!" قام تراسيماكوس عن كرسيه. "كل شيء يحتاج المترجم أن يعرفه موجود في الشفرة! قد تناقش أيضاً أن كل شيء تحتاجين أن تعرفيه موجود في الملف التنفيذي الثنائي! إن كنت ذكية كفاية لتقرئيه! ما *تنوي* أن تقوم به ليس في الشفرة."

أدرك تراسيماكوس أنه كان واقفاً وجلس. "سقراط، هذا أمر تافه، لمَ علينا أن نتناقش عن كون التعليقات قيّمة أو لا؟ كل شيء قرأته بحياتي يقول إنها قيّمة وينبغي أن تُستخدم بتحرر. نحن نضيّع الوقت."

"اهدأ، تراسيماكوس، واسأل كاليكليس مذ متى بدأ يبرمج."[1]

"مذ متى، كاليكليس؟"

"حسناً، بدأت في أكروبوليس 4 من 15 سنة مضت. أظن أنني رأيت دزينة من الأنظمة الرئيسة من اللحظة التي ولدت فيها إلى الوقت الذي أعطيناها فيها فنجاناً من الشوكران (نبات سام يستخدم لتسكين الآلام). وقد عملت في أجزاء رئيسة من دزينات أخرى. اثنان منها كان كل واحد منهما نظاماً فيه أكثر من نصف مليون سطر من الشفرة، لذا أنا أعلم عمَّ أتكلم. التعليقات عديمة الفائدة تماماً."

نظر سقراط إلى المبرمج الأصغر. "كما قال كاليكليس، لدى التعليقات العديد من المشاكل القانونية، ولن تدرك ذلك بدون المزيد من الخبرة. إن لم تُكتب بطريقة صحيحة، فإنها أسوأ من عدم النفع."

"حتى وإن كُتبت بطريقة صحيحة، فهي عديمة النفع،" قال كاليكليس. "التعليقات أقل دقة من لغة البرمجة. أنا أفضل ألا يكونوا لدّي مطلقاً."

"انتظر دقيقة،" قال سقراط. "توافق إزمين على أن التعليقات أقل دقة. لكن فكرتها هي أن التعليقات تعطيك مستوى أعلى من التجريد، ونحن نعلم أن تلك المستويات من التجريد هي واحدة من أقوى أدوات المبرمج."

"لا أوافق على هذا،" أجاب غلاكون. "بدلاً من التركيز على كتابة التعليقات، ينبغي أن تركز على جعل الشفرة مقروءة أكثر. تزيل إعادة التصنيع معظم تعليقاتي. حالما أعيد التصنيع، قد يكون لدى شفرتي 20 أو 30 استدعاء لإجرائية دون الحاجة لأي تعليقات. يستطيع المبرمج الجيد أن يقرأ المقصد من الشفرة نفسها، وما الجيد من قراءة مقصد أحدهم عندما تعلم أن الشفرة تمتلك خطأ؟" غلاكون كان سعيداً بمساهمته، وكاليكليس هز برأسه هزة الموافقة.

"يبدو أنكم يا شباب ولا مرة توجّب عليكم أن تعدلوا شفرة شخص آخر،" قالت إزمين. فجأة بدا كاليكليس مهتماً جداً بعلامات قلم الرصاص على قرميدات السقف. "لم لا تحاولان قراءة شفرتكما بعد ستة أشهر أو سنة من كتابتها؟ تستطيعان أن تحسنا قدرة قراءة الشفرة وكتابة التعليقات لديكما. لا يجب عليكما أن تختارا واحداً أو الآخر. إن كنت تقرأ رواية، قد لا تريد عناوين الأقسام. لكن إن كنت تقرأ كتاباً تقنياً، ستفضل أن تكون قادراً على إيجاد ما تبحث عنه بسرعة. لا ينبغي أن انتقل إلى نظام التركيز الفائق وأقرأ مئات الأسطر من الشفرة فقط كي أجد السطرين الذين أريد أن أغيرهما."

[1] كما هو واضح، في مستوىً ما يجب أن تكون التعليقات مفيدة. أن تصدّق عكس ذلك يعني أن تصدّق أن فهم البرنامج مستقل عن كمية المعلومات التي عرفها القارئ مسبقاً عنه. -بي. إيه. شيل

"حسناً، أستطيع أن أرى أنه من المفيد أن نكون قادرين على مسح الشفرة بنظرة،" قال غلاكون. لقد رأى بعضاً من برامج إزمين وكان معجباً بها. "لكن ماذا عن فكرة كاليكليس الأخرى، أن التعليقات تصبح منتهية الصلاحية عند تغير الشفرة؟ فقد بدأت البرمجة من عدة سنين فقط، ومع ذلك أعلم أن لا أحد يحدّث تعليقاته."

"حسناً، نعم ولا،" قالت إزمين. "إن اعتبرت أن التعليق مضمون وأن الشفرة مشتبه بها، فإنك في مشكلة كبيرة. فعلياً، إيجاد عدم توافق بين التعليق والشفرة يميل ليعني أن كلاهما خاطئ. حقيقة أن بعض التعليقات سيئة لا تعني أن كتابة التعليقات سيئة. سأذهب إلى غرفة الغداء لآتي بوعاء آخر من القهوة." غادرت إزمين الغرفة.

"اعتراضي الرئيس على التعليقات،" قال كاليكليس، "هو أنها مضيعة للموارد."

"هل يستطيع أحد ما أن يفكر بطرق لتخفيض الوقت الذي تستهلكه كتابة التعليقات؟" سأل سقراط.

"تصميم الإجرائيات بالشفرة الزائفة، وبعدها تحويل الشفرة الزائفة إلى تعليقات وملء الشفرة بينها،" قال غلاكون.

"حسناً، هذا سيعمل طالما أن التعليقات لا تكرر الشفرة،" قال كاليكليس.

"كتابة تعليق يجعلك تفكر بجد أكثر عما تقوم به شفرتك،" قالت إزمين وهي عائدة من غرفة الغداء. "إن كان من الصعب أن تكتب تعليقاً، فإما أن تكون الشفرة سيئة أو أن تكون أنت لمّا تفهمها بشكل جيد كفاية. بكلا الحالين. تحتاج أن تصرف المزيد من الوقت على الشفرة، لذا فالوقت الذي تصرفه في كتابة التعليقات ليس ضائعاً لأنه نبهك إلى عمل لازم."

"حسناً،" قال سقراط. "لا أستطيع أن أفكر بالمزيد من الأسئلة، وأعتقد أن إزمين غلبتكم اليوم يا شباب. سنشجع كتابة التعليقات، لكن لن نكون ساذجين معها. سيكون لدينا مراجعات للشفرة بحيث سيحصل كل واحد على فكرة جيدة عن نوع التعليقات التي تساعد فعلياً. إن كان لديكم مشاكل في فهم شفرة شخص آخر، دعوه يعرف كيف يستطيع أن يحسّنها."

32. 4 مفاتيح التعليقات الفعالة

بماذا تقوم الإجرائية التالية؟[1]

[1] طالما يوجد أهداف مُعرَّفة بشكل خاطئ، وثغرات شاذة، ومواعيد غير واقعية، سيكون هناك مبرمجون راغبون بالقفز إلى وحلّ المشكلة، موفرين التوثيق إلى ما بعد. حياة طويلة فورتران! --إد بوست

إجرائية جافا الغامضة رقم 1

```
// write out the sums 1..n for all n from 1 to num
current = 1;
previous = 0;
sum = 1;
for ( int i = 0; i < num; i++ ) {
   System.out.println( "Sum = " + sum );
   sum = current + previous;
   previous = current;
   current = sum;
}
```

تخمينك الأفضل؟

هذه الإجرائية تحسب أول *num* رقم من متتالية فيبوناتشي. أسلوبها في كتابة الشفرة أفضل بقليل من أسلوب الإجرائية في بداية الفصل، لكن التعليق خاطئ، وإن وثقت بعمى بالتعليق، تكون قد نزلت في مسار الوعود بالاتجاه الخاطئ.

ماذا عن هذه؟

إجرائية جافا الغامضة رقم 2

```
// set product to "base"
product = base;

// loop from 2 to "num"
for ( int i = 2; i <= num; i++ ) {
   // multiply "base" by "product"
   product = product * base;
}
System.out.println( "Product = " + product );
```

ترفع الإجرائية العدد الصحيح *base* إلى القوة *num*. التعليقات في هذه الإجرائية دقيقة، لكنها لا تضيف أي شيء إلى الشفرة. إنها مجرد إصدار مسهب أكثر من الشفرة نفسها.

إليك آخر إجرائية:

إجرائية جافا الغامضة رقم 3

```
// compute the square root of Num using the Newton-Raphson approxima-
tion
r = num / 2;
while ( abs( r - (num/r) ) > TOLERANCE ) {
   r = 0.5 * ( r + (num/r) );
}
System.out.println( "r = " + r );
```

تحسب هذه الإجرائية الجذر التربيعي ل *num*. الشفرة ليست عظيمة، لكن التعليق دقيق.

أي الإجرائيات كان اكتشافها أسهل عليك؟ ولا واحدة من الإجرائيات مكتوبة بشكل جيد بشكل خاص-أسماء المتحولات بشكل خاص رديئة. بإيجاز، على كل، توضح هذه الإجرائيات قوة وضعف التعليقات الداخلية. لدى الإجرائية رقم 1 تعليقات غير دقيقة. تعليقات الإجرائية رقم 2 هي مجرد تكرار للشفرة ولذلك فهي عديمة النفع.

فقط تعليقات الإجرائية رقم 3 تكسب أجرتها. وجود تعليقات رديئة أسوأ من ألا توجد تعليقات. الإجرائية 1 و2 ستكونان أفضل بدون تعليقات منهما مع تعليقات رديئة مثل التي امتلكاها.

يصف القسم التالي مفاتيح كتابة تعليقات فعالة.

أنواع التعليقات

يمكن أن تُصنَّف التعليقات إلى ست فئات:

تكرار الشفرة

التعليقات المكررة تعيد التعبير عما تفعله الشفرة بكلمات مختلفة. فهي، فقط، تعطي قارئ الشفرة المزيد ليقرأه بدون أن تقدم معلومات إضافية.

شرح الشفرة

تُستخدم التعليقات التوضيحية عادة لتشرح قطع الشفرة المعقدة أو المخادعة أو الحساسة. في هكذا حالات هي مفيدة، لكن يكون ذلك عادة لأن الشفرة مربكة. إن كانت الشفرة معقدة جداً بحيث تحتاج إلى أن تُشرح، فتقريباً دائماً من الأفضل أن تُحسِّن الشفرة بدلاً من أن تضيف تعليقات. اجعل الشفرة نفسها أوضح، ثم استخدم تعليقات القصد أو التلخيص.

علامات في الشفرة

تعليق العلامة هو تعليق لا يُنوى على إبقائه في الشفرة. إنه ملاحظة للمطور أن العمل لم ينجز بعد. يكتب بعض المطورين علامات خاطئة قواعدياً (******، على سبيل المثال) لذا يلوح المترجم إليها ويذكرهم أن لديهم المزيد من العمل ليقوموا به. يضع مطورون آخرون مجموعة مخصصة من المحارف في التعليقات لا تتعارض مع الترجمة بحيث يستطيعون البحث عنها.

بعض المشاعر أسوأ من أن تحصل على تقرير من الزبون عن مشكلة في الشفرة، *وتصحيح* المشكلة، وتتبعها إلى قسم من الشفرة حيث تجد شيءاً ما مثل:

```
return NULL; // ****** NOT DONE! FIX BEFORE RELEASE!!!
```

إصدار شفرة مختلّة للزبون أمر سيئ كفاية؛ وإصدار شفرة علمتَ أنها مختلة أمر أسوأ حتى.

لقد وجدت أن معيرة أسلوب تعليقات العلامات مفيد. إن لم تمعير، سيستخدم بعض المبرمجين ******، وبعضهم !!!!!!، وبعضهم *TBD*، وبعضهم أعرافاً أخرى متنوعة. يجعل استخدام تشكيلة من التدوينات البحث الآلي عن الشفرة غير المكتملة ميالاً للخطأ أو مستحيلاً. المعيرة وفق أسلوب علامة محددة يسمح لك أن تقوم بالبحث

الآلي عن الأقسام غير المكتملة من الشفرة كإحدى خطوات لائحة اختبار الإصدار، ما يمنع مشكلة *"أصلح قبل الإصدار!!!" FIX BEFORE RELEASE!!!*. تدعم بعض المحررات لاحقات "to do (tags)" وتسمح لك بالانتقال إليها بسهولة.

تلخيص الشفرة

التعليق الذي يلخّص الشفرة يفعل الآتي تماماً: يركّز عدة سطور من الشفرة في جملة أو جملتين. هكذا تعليقات أكثر قيمة من التعليقات التي تكرر الشفرة فقط لأن القارئ يستطيع أن يمسحها بنظرة بسرعة أكبر من الشفرة. تعليقات التلخيص مفيدة بشكل خاص عندما يحاول شخص ما غير كاتب الشفرة الأصلي أن يعدّل الشفرة.

وصف قصد الشفرة

التعليق في هذا المستوى يشرح الغرض من قسم من الشفرة. تعمل تعليقات القصد في مستوى المشكلة أكثر من مستوى الحل. مثلاً،

```
-- get current employee information
```

هو تعليق قصد، بينما

```
-- update employeeRecord object
```

هو تعليق تلخيص بمفردات الحل. وجدت دراسة استمرت لستة شهور بقيادة **آي بي إم** أن مبرمجي الصيانة "كانوا يقولون غالباً أن فهم قصد المبرمج الأصلي كان المشكلة الأصعب" (فجيلسفاد وهاملين 1979). التفريق بين تعليقات القصد والتلخيص لا يكون واضحاً دوماً، وعادةً لا يكون مهماً. تُطرح أمثلة عن تعليقات القصد خلال هذا الفصل.

المعلومات التي لا يمكن أن يُعبَّر عنها بالشفرة نفسها

بعض المعلومات لا يمكن التعبير عنها بالشفرة لكن يتحتم أن تبقى في الشفرة المصدرية. هذه الفئة من التعليقات تتضمن ملاحظات حقوق النسخ، وملاحظات الخصوصية، وأرقام الإصدارات، وتفاصيل أخرى لتدبير الأمور، وملاحظات عن تصميم الشفرة؛ ووصلات إلى متطلبات أو توثيق هيكلة ذوي صلة؛ ومؤشرات إلى مراجع على الشبكة، وملاحظات المعايرة؛ والتعليقات اللازمة لأدوات التنسيق مثل **Javadoc** و **Doxygen**؛ وهلم جراً.

الأنواع الثلاثة من التعليقات المقبولة للشفرة المكتملة هي: المعلومات التي لا يمكن أن يُعبَّر عنها بالشفرة، وتعليقات القصد، وتعليقات التلخيص.

فعالية التعليق

ليست كتابة التعليقات ذلك المستهلك للوقت. الكثير جداً من التعليقات بمقدار سوء القليل جداً، وتستطيع أن تنجز أرضية متوسطة بشكل اقتصادي.

يمكن أن تستهلك التعليقات الكثير من الوقت لتُكتب لسببين شائعين. أولاً، قد يكون أسلوب كتابة التعليقات مستهلكاً للوقت أو مملاً. إن كان كذلك، جد أسلوباً آخر. أسلوب كتابة التعليقات الذي يفرض الكثير من العمل المربك هو وجع رأس في الصيانة. إن كانت التعليقات صعبة التغيير، لن تُغيّر وستصبح غير دقيقة ومضللة، ما هو أسوأ من ألا تكون تعليقات من الأساس.

ثانياً، قد تكون كتابة التعليقات صعبة لأن الكلمات التي تصف ما يقوم به البرنامج لا تأتي بسهولة. عادةً، هذه إشارة إلى أنك لا تفهم ما يقوم به البرنامج. الوقت الذي تصرفه في "التعليق" هو فعلياً وقت تصرفه في فهم أفضل للبرنامج، والذي هو وقت تحتاج أن تصرفه بغض النظر عن إن كنت تكتب تعليقاً أو لا.

فيما يلي توجيهات لكتابة تعليقات بطريقة فعالة:

استخدم أساليباً لا تفرمل أو تثبط التعديلات أي أسلوب مزخرف جداً هو أسلوب مزعج في الصيانة. مثالاً، أشّر على الأجزاء من التعليق في الأدنى التي لن تُصان:

مثال جافا عن أسلوب كتابة تعليقات صعب الصيانة

```
// Variable        Meaning
// --------        -------
// xPos .......... XCoordinate Position (in meters)
// yPos .......... YCoordinate Position (in meters)
// ndsCmptng...... Needs Computing (= 0 if no computation is needed,
//                                 = 1 if computation is needed)
// ptGrdTtl....... Point Grand Total
// ptValMax....... Point Value Maximum
// psblScrMax..... Possible Score Maximum
```

إن قلت إن نقاط التوجيه (.....) ستكون صعبة الصيانة، فأنت محق! إنها تبدو جميلة، لكن اللائحة جيدة بدونها. إنها تضيف عملاً مربكاً إلى وظيفة تعديل التعليقات، وستفضل أن يكون لديك تعليقات دقيقة بدلاً من أخرى حسنة المنظر، إن كان هذا هو اختيارك-وعادةً يكون.

هاهنا مثال آخر على أسلوب شائع صعب الصيانة:

مثال سي ++ عن أسلوب كتابة تعليقات صعب الصيانة

```
/***********************************************************************
 * class: GigaTron (GIGATRON.CPP)                                      *
 *                                                                     *
 * author: Dwight K. Coder                                             *
 * date: July 4, 2014                                                  *
 *                                                                     *
 * Routines to control the twenty-first century's code evaluation      *
 * tool. The entry point to these routines is the EvaluateCode()       *
 * routine at the bottom of this file.                                 *
 ***********************************************************************/
```

هذه كتلة تعليقات حسنة المنظر. من الواضح أن كل الكتلة تنتمي إلى بعضها، وبداية ونهاية الكتلة بيّنتين. ما هو غير واضح حول هذه الكتلة هو مقدار سهولة تغييرها. إن كان عليك أن تضيف اسماً لملف في أسفل التعليق، فالفرص تماماً جيدة بأنك ستثار من العمود الجميل من النجوم في اليمين. إن احتجت أن تغيّر تعليقات الفقرة، ستثار من النجوم في اليمين واليسار. في التطبيق، هذا يعني أن الكتلة لن تُعدّل لأنه سيكون عملاً كثيراً جداً. إن استطعت أن تكبس زراً وتحصل على عمود مهندم من النجوم، فإنه عظيم. استخدمه. ليست المشكلة بالنجوم بل في أن النجوم صعبة الصيانة. يبدو التعليق التالي تقريباً بجودة السابق وهو أسهل صيانةً بكثير:

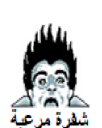
مثال سي ++ عن أسلوب كتابة تعليقات سهل الصيانة

```
/**********************************************************************
   class: GigaTron (GIGATRON.CPP)

   author: Dwight K. Coder
   date: July 4, 2014

   Routines to control the twenty-first century's code evaluation
   tool. The entry point to these routines is the EvaluateCode()
   routine at the bottom of this file.
**********************************************************************/
```

هاهنا أسلوب صعب الصيانة بشكل خاص:



مثال فيجوال بيسك عن أسلوب تعليق صعب الصيانة

```
'  set up Color enumerated type
'  +--------------------------+
   ...

'  set up Vegetable enumerated type
'  +------------------------------+
   ...
```

من الصعب أن تعرف ما القيمة التي تضيفها إشارة الزائد في بداية ونهاية كل سطر من الشحطات إلى التعليق، لكن من السهل أن تخمن أنه في كل مرة يتغير التعليق، يجب أن يُعاير السطر الذي تحته بحيث تكون إشارة الزائد في النهاية في المكان الصحيح بدقة. وما ذا تفعل إذا فاض التعليق على سطرين؟ كيف ستحاذي إشارات الزائد؟ أتنزع كلمات من التعليق حتى يستهلك سطراً واحداً؟ هل تجعل السطرين بنفس الطول؟ المشاكل في هذا النهج تتضاعف عندما تحاول أن تطبقه بشكل متماسك.

تظهر توجيهات عامة لـ**جافا و سي** ++ من دافع مشابه، بأن تستخدم تركيب ال// لتعليقات السطر الواحد و تركيب ال/* ... */ للتعليقات الأطول، كما يظهر هنا:

مثال جافا عن استخدام تركيبي تعليق لغرضين مختلفين

```
// This is a short comment
...
/* This is a much longer comment. Four score and seven years ago our
fathers brought forth on this continent a new nation, conceived in
liberty and dedicated to the proposition that all men are created
equal. Now we are engaged in a great civil war, testing whether that
nation or any nation so conceived and so dedicated can long endure.
We are met on a great battlefield of that war. We have come to dedi-
cate a portion of that field as a final resting-place for those who
here gave their lives that that nation might live. It is altogether
fitting and proper that we should do this.
*/
```

التعليق الأول سهل الصيانة طالما هو قصير. من أجل التعليقات الأطول، مهمة إنشاء أعمدة طويلة من الشرطات "/" الثنائية، تُجزّء بشكل آلي خطوطاً من النص بين السطور، وهكذا نشاطات لا تكافئ الكثير، لذا تركيب ال/* ... */ أنسب للتعليقات متعددة الأسطر.

نقطة مفتاحية

النقطة هنا أنه يجب أن تنتبه إلى كيفية صرفك لوقتك. إن صرفت الكثير من الوقت في إدخال وحذف الشحطات لتجعل علامات الزائد منتظمة، فأنت لا تبرمج؛ أنت تضيع الوقت. جد أسلوباً أكثر فعالية. في حالة التسطير وعلامات الزائد، بإمكانك أن تختار أن تمتلك التعليق فقط بدون أي تسطير. إن احتجت أن تستخدم التسطير للتأكيد، جد طريقة ما غير التسطير وعلامات الزائد لتؤكد على هذه التعليقات. إحدى الطرق أن تمتلك تسطيراً معيارياً يكون دائماً بنفس الطول مهما كان طول التعليق. هكذا سطر لا يفرض أي صيانة، وبإمكانك استخدام ماكرو برنامج لتنسيق النص لتدخله في المقام الأول.

استخدم عملية البرمجة بالشفرة الزائفة[1] إن رسمت هيكل الشفرة بالتعليقات قبل أن تكتبها، فإنك تفوز بعدة طرق. عندما تُنهي الشفرة، تكون قد أنجزت التعليقات. لا يجب عليك أن تكرّس وقتاً للتعليق. وأنت أيضاً تكسب كل المنافع التصميمية من الكتابة بالشفرة الزائفة عالية المستوى قبل أن تملأ شفرة لغة البرمجة منخفضة المستوى.

أدمج كتابة التعليقات بأسلوبك في التطوير بديل أن تدمج كتابة التعليقات بأسلوبك في التطوير هو أن تترك كتابة التعليقات حتى نهاية المشروع، وهذا فيه الكثير جداً من السلبيات. إنها تصبح عملاً بمجهودها الذاتي، ما يجعلها تبدو كعمل واجب أكثر مما لو أنجزت شيئاً فشيئاً. تستغرق كتابة التعليقات لاحقاً وقتاً أكثر لوجوب أن تذكر أو تكتشف ما تفعله الشفرة بدلاً من مجرد كتابة ما تفكر به حالياً. وهي أيضاً أقل دقة لأنك تميل لتنسى افتراضات أو دقائق في التصميم.

[1] **إشارة مرجعية** لتفاصيل عن عملية البرمجة بالشفرة الزائفة، انظر الفصل 9، "عملية البرمجة بالشفرة الزائفة."

الحجة الشائعة ضد التعليق وأنت تتقدم هي "عندما تكون مركزاً في الشفرة، لا ينبغي أن تكسر تركيزك كي تكتب تعليقات." الاستجابة المناسبة هي أنه، إن وجب عليك أن تركز بجدّ كبير في كتابة الشفرة، وكتابة التعليقات تقاطع تفكيرك، فأنت تحتاج أن تصمم بالشفرة الزائفة أولاً ثم تحوّل الشفرة الزائفة إلى تعليقات. الشفرة التي تفرض هكذا تركيز شديد هي إشارة تحذير.



إن كان تصميمك صعب أن يُكتب شفرةً، بسّط التصميم قبل أن تهتم بالتعليقات أو الشفرة. إن استخدمت الشفرة الزائفة لتوضّح أفكارك، ستكون كتابة الشفرة عملية مستقيمة والتعليقات ستكون آلية.

الأداء ليس سببا جيداً لتتجنب كتابة التعليقات إحدى الصفات الدورية لموجة التقنية المتدحرجة نوقشت في القسم 4. 3، "موقعك على الموجة التقنية،" هل تفرض كتابة التعليقات في البيئات التفسيرية ضريبة أدائية قابلة للقياس. في ثمانينات القرن الماضي، التعليقات في برامج **بيسك** على حواسيب **آي بي إم** الأصلية أبطأت البرامج. في تسعينيات القرن الماضي، صفحات *asp.* قامت بنفس الأمر. في العقد الأول من هذا القرن شفرة **جافا سكريبت** والشفرات الأخرى التي تحتاج أن تُبعَث عبر اتصالات الشبكة قدمت مشكلة مشابهة.

في كل الحالات هذه، الحل النهائي لم يكن تجنب كتابة التعليقات؛ بل كان إنشاء إصدار من الشفرة مختلف عن إصدار التطوير. وأنجز هذا بتشغيل الشفرة نمطيّاً عبر أداة تزيل التعليقات كجزء من عملية البناء.

العدد المثالي للتعليقات

أشار كابيرز جونز إلى أن الدراسات في **آي بي إم** وجدت أن كثافة التعليقات بتعليق واحد تقريباً كل 10 عبارات كانت هي الكثافة التي بدى فيها الوضوح في القمة. جعلت التعليقات الأقل الشفرة صعبة الفهم. وخفّضت التعليقات الأكثر من قابلية فهم الشفرة أيضاً (جونز 2000).

يمكن أن يساء استخدام هذا النوع من الأبحاث، وأحياناً تتبنى المشاريع معياراً مثل "يتحتم على البرامج أن تحتوي تعليقاً واحداً كل خمسة سطور." يعالج هذا المعيار عَرَض عدم كتابة المبرمجين لشفرة واضحة، لكنه لا يعالج السبب.

إن استخدمت عملية البرمجة بالشفرة الزائفة بشكل فعال، قد تنتهي بتعليق لكل عدة سطور من الشفرة. عدد التعليقات، على كل، سيكون أثراً جانبياً للعملية نفسها. بدلاً من التركيز على عدد التعليقات، ركز على إن كان كل تعليق فعالاً أو لا. إن وصفت التعليقات لمَ كُتبت الشفرة وطابقت المعايير الأخرى المؤكدة في هذا الفصل، سيكون لديك تعليقات كافية.

32. 5 تقنيات كتابة التعليقات المراجعة إلى هنا

تكون التعليقات سهلة التعديل بعدة تقنيات مختلفة لكن حسب المستوى الذي تُتطبق التعليقات فيه: البرنامج، والملفات، والإجرائيات، والفقرات، والسطور المفردة.

التعليق على سطور مفردة

في الشفرة الجيدة، الحاجة إلى التعليق على سطور مفردة من الشفرة أمر نادر. ها هنا سببين محتملين لحاجة سطر من الشفرة إلى تعليق:

- السطر المفرد معقد كفاية حتى يحتاج إلى شرح.
- احتوى هذا السطر المفرد مرة على خطأ، وأنت تريد أن تسجل هذا الخطأ.

إليك بعض التوجيهات للتعليق على سطر مفرد من الشفرة:

تجنب التعليقات المنغمسة في الذات منذ العديد من السنين الماضية، سمعت قصة عن مبرمج صيانة أوقظ من الفراش كي يصلح برنامج مختلاً. المبرمج الذي كتب البرنامج غادر الشركة ولا يمكن الوصول إليه. ومبرمج الصيانة لم يعمل على البرنامج من قبل، وبعض فحص التوثيق بحذر، وجد تعليقاً واحداً فقط، كان يبدو كهذا:

```
MOV AX, 723h     ; R. I. P. L. V. B.
```

بعد العمل على البرنامج خلال الليل والحيرة من التعليق، قام المبرمج بإصلاح ناجح وذهب إلى بيته إلى الفراش. بعد شهور، التقى بكاتب البرنامج في مؤتمر واكتشف أن التعليق كان يمثل: "ارقد بسلام، لدويدج فان بيتهووفين" "Rest in peace, Ludwig van Beethoven". مات بيتهووفين في 1827 (في النظام العشري)، والذي هو 723 (في النظام الست عشري). حقيقة أن h723 لزمت في تلك البقعة ليس لها علاقة بالتعليق. آآآآررررررهههههه!

تعليقات نهاية السطر ومشاكلها

تعليقات نهاية السطر هي التعليقات التي تظهر في نهايات سطور الشفرة:

مثال فيجوال بيسك عن تعليقات نهاية السطر

```
For employeeId = 1 To employeeCount
   GetBonus( employeeId, employeeType, bonusAmount )
   If employeeType = EmployeeType_Manager Then
      PayManagerBonus( employeeId, bonusAmount ) ' pay full amount
   Else
      If employeeType = EmployeeType_Programmer Then
         If bonusAmount >= MANAGER_APPROVAL_LEVEL Then
            PayProgrammerBonus( employeeId, StdAmt() ) ' pay std. amount
         Else
            PayProgrammerBonus( employeeId, bonusAmount ) ' pay full amount
         End If
      End If
   End If
Next
```

مع أنه مفيد في بعض الحالات، يخلق تعليق نهاية السطر عدة مشاكل. يجب على التعليقات أن تُحاذى إلى يمين الشفرة بحيث لا تتداخل مع البنيان المرئي للشفرة. إن لم تُحاذها على نحو مرتب، ستجعل تعدادك (نص شفرتك) يبدو كأنه خرج من غسالة. تميل تعليقات نهاية السطر لتكون صعبة التنسيق. إن استخدمت العديد منها، ستستغرق وقتاً لتحاذها. هكذا وقت غير مصروف في تعلم المزيد عن الشفرة، إنه مكرس فقط لمهمة الترتيب من كبس لمفتاحي المسطرة (space)والجدولة (tab)

تميل تعليقات نهاية السطر أيضاً لتكون مبهمة. الناحية اليمنى من السطر لا تقدم عادة مساحة كافية، ورغبة الحفاظ على التعليق على سطر واحد تعني حتمية كون التعليق قصيراً. فيذهب العمل لجعل السطر قصيراً قدر الإمكان وليس واضحاً قدر الإمكان.

تجنب تعليقات نهاية السطر للسطور المفردة بالإضافة إلى مشاكلها التطبيقية، تخلق تعليقات نهاية السطر عدة مشاكل فهمية. إليك مثال عن مجموعة من تعليقات نهاية السطر:

مثال سي ++ عن تعليقات نهاية السطر عديمة الفائدة

```
memoryToInitialize = MemoryAvailable();    // get amount of memory available
pointer = GetMemory( memoryToInitialize ); // get a ptr to the available memory
ZeroMemory( pointer, memoryToInitialize ); // set memory to 0
...
FreeMemory( pointer );                     // free memory allocated
```

تقوم التعليقات فقط بتكرار الشفرة

صعوبة كتابة تعليق ذي معنى لسطر واحد من الشفرة هي مشكلة متعلقة بالنظام في تعليقات نهاية السطر. معظم تعليقات نهاية السطر فقط تكرر سطر الشفرة الذي تؤذيه أكثر مما تساعده.

تجنب تعليقات نهاية السطر لسطور متعددة من الشفرة إن قًصد تطبيق تعليق نهاية السطر على أكثر من سطر واحد من الشفرة، لن يظهر التنسيق أي السطور يُطبَّق عليها التعليق:

شفرة مرعبة

مثال فيجوال بيسك عن تعليق نهاية سطر مربك على عدة سطور من الشفرة

```
For rateIdx = 1 to rateCount                  ' Compute discounted rates
   LookupRegularRate( rateIdx, regularRate )
   rate( rateIdx ) = regularRate * discount( rateIdx )
Next
```

مع أن محتوى هذا التعليق جيد، فمكانه ليس كذلك. عليك أن تقرأ التعليق والشفرة لتعرف إن كان التعليق يُطبَّق على عبارة محددة أو على كامل الحلقة.

متى تُستخدم تعليقات نهاية السطر

ضع في بالك ثلاثة استثناءات لتوصيات عدم استخدام تعليقات نهاية السطر:

استخدم تعليقات نهاية السطر لتعلق على التصريحات عن البيانات [1] تعليقات نهاية السطر مفيدة في التعليق على التصريحات عن البيانات لأنها لا تمتلك نفس المشاكل المتعلقة بالنظام مثل تعليقات نهاية السطر على الشفرة، شريطة أن تمتلك عرضاً كافياً. بوجود 132 عمود، تستطيع عادةً أن تكتب تعليقاً ذا معنى بجانب كل تصريح عن بيانات:

مثال جافا عن تعليقات نهاية سطر جيدة على تصريحات عن بيانات
```
int boundary = 0;           // upper index of sorted part of array
String insertVal = BLANK; // data elmt to insert in sorted part of array
int insertPos = 0;          // position to insert elmt in sorted part of array
```

تجنب استخدام تعليقات نهاية السطر من أجل ملاحظات الصيانة تُستخدم تعليقات نهاية السطر أحياناً لتسجيل التعديلات على الشفرة بعد تطويرها الأولي. يتألف هذا النوع من التعليقات عادةً من بيانات والحروف الأولى من اسم المبرمج، أو ربما رقم تقرير الخطأ. هاهنا مثال:

```
for I = 1 to maxElmts – 1     -- fixed error #A423 10/1/05 (scm)
```

إضافة هكذا تعليق يمكن أن يكون مرضياً بعد جلسة *تصحيح* في سهرة متأخرة على برمجية قيد الإنتاج، لكن لا مكان فعلياً لهكذا تعليقات في شفرة الإنتاج. تُعامل هكذا تعليقات بشكل أفضل عن طريق برمجية تحكم بالإصدار. ينبغي أن تشرح التعليقات لمَ تعمل الشفرة *الآن*، ليس لمَ لم تعمل الشفرة في فترة ما في الماضي.

استخدم تعليقات نهاية السطر لتعلّم نهايات الكتل [2] يكون تعليق نهاية السطر مفيداً لتعليم نهاية كتلة طويلة من الشفرة-نهاية حلقة `while` أو عبارة `if`، مثلاً. هذا مشروح بتفصيل أكثر لاحقاً في هذا الفصل.

بجانب زوج من الحالات الخاصة، تمتلك تعليقات نهاية السطر مشاكل فهمية وتميل لتُستخدم من أجل الشفرة الشديدة التعقيد. هي أيضاً صعبة التنسيق والصيانة. بعد كل شيء، من الأفضل تجنبها.

التعليقات على فقرات من الشفرة

معظم التعليقات في البرامج الموثقة بشكل جيد هي جملة واحدة أو جملتين من التعليقات تصف مقطعاً من الشفرة:

مثال جافا عن تعليقات جيدة على فقرة من الشفرة
```
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

[1] **إشارة مرجعية** الجوانب الأخرى من تعليقات نهاية السطر على التصريحات عن البيانات موصوفة في " التعليق على التصريحات عن البيانات،" لاحقاً في هذا الفصل.

[2] **إشارة مرجعية** استخدام تعليقات نهاية السطر لتعلّم نهايات الكتل موصوف بعمق أكثر في "التعليق على بنى التحكم،" لاحقاً في هذا الفصل.

لا يكرر التعليق الشفرة -إنه يصف القصد من الشفرة. تكون هكذا تعليقات نسبياً سهلة الصيانة. حتى وإن وجدت خطأً في الطريقة التي تُبدل فيها الجذور، مثلاً، فلن يلزم تغيير التعليق. تكون التعليقات غير المكتوبة في مستوى القصد أصعب في الصيانة.

اكتب التعليقات في مستوى القصد من الشفرة صف الغرض من كتلة الشفرة التي تلي التعليق. هاهنا مثال على تعليق غير فعال لأنه لا يعمل في مستوى القصد[1]:

```
مثال جافا عن تعليق غير فعال
/* check each character in "inputString" until a dollar sign
is found or all characters have been checked
*/
done = false;
maxLen = inputString.length();
i = 0;
while ( !done && ( i < maxLen ) ) {
   if ( inputString[ i ] == '$' ) {
      done = true;
   }
   else {
      i++;
   }
}
```

تستطيع أن تكتشف أن الحلقة تبحث عن $ بقراءة الشفرة، ونوعاً ما يكون تلخيص ذلك في تعليق مفيداً. المشكلة في هذا التعليق أنه يكرر الشفرة فقط ولا يقدم أي رؤية عما يُفترض أن تقوم به الشفرة. سيكون هذا التعليق أفضل بقليل:

```
// find '$' in inputString
```

هذا التعليق أفضل لأنه يحدد أن هدف الحلقة هو إيجاد $. لكنه لا يقدم أي رؤية إضافية عن سبب حاجة الحلقة إلى إيجاد $-بكلمات أخر عن القصد الأعمق من الحلقة. هاهنا مثال لا يزال أفضل:

```
// find the command-word terminator ($)
```

يحتوي هذا التعليق فعلياً على معلومة، تعداد الشفرة لا يحتويها، أعني ال $ تنهي "كلمة أمر". لا تستطيع بأية طريقة أن تخمن تلك الحقيقة بمجرد قراءة مقطع الشفرة، لذا فإن التعليق نافع بإخلاص.

طريقة أخرى للتفكير بكتابة التعليقات في مستوى القصد هي أن تفكر بمَ ستسمي إجرائية تقوم بنفس الشيء الذي تقوم به الشفرة التي تريد التعليق عليها. إن كنت تكتب فقرات من الشفرة لكل منها غرض واحد، فلن يكون ذلك صعباً. التعليق في مثال الشفرة السابق هو مثال جيد. *FindCommandWordTerminator()* سيكون اسماً لائقاً لإجرائية. الخيار الآخر *Find$InInputString()* و *CheckEachCharacterInInputStrUntilADollarSignIsFoundOrAllCharactersHaveBeenChecked()*

[1] **إشارة مرجعية** هذه الشفرة التي تنجز بحث سلسلة نصية بسيطاً تُستخدم فقط لأغراض الشرح. من أجل شفرة حقيقية، قد تستخدم توابع مكتبة السلاسل النصية المدمجة في **جافا** بدلاً. للمزيد عن أهمية فهم مقدرات لغتك، انظر "اقرأ" في القسم 33.3

هما اسمان رديئان (أو غير صالحان) لأسباب بينة. اكتب الوصف بدون تقصيره أو اختصاره، كما يمكن أن تفعل لاسم إجرائية. هذا الوصف هو تعليقك، ومن المحتمل أن يكون في مستوى القصد.



ركز جهود توثيقك على الشفرة نفسها بالأرقام القياسية، الشفرة نفسها هي دائماً التوثيق الأول الذي ينبغي أن تفحصه. في المثال السابق، الحَرفيّة $ ينبغي أن تُبدّل بثابت مسمى وينبغي أن يقدم اسم المتحول أفكاراً أكثر عمَّ يجري. إن أردت أن تدفع حافة غلاف قابلية القراءة، أضف متحولاً ليحتوي نتيجة البحث. القيام بذلك يميّز بوضوح بين دليل الحلقة ونتيجة الحلقة. هاهنا الشفرة المعادة كتابتها مع تعليقات جيدة وأسلوب جيد:

مثال جافا عن تعليق جيد و شفرة جيدة

```
// find the command-word terminator
foundTheTerminator = false;
commandStringLength = inputString.length();
testCharPosition = 0;
while ( !foundTheTerminator && ( testCharPosition < commandStringLength ) ) {
   if ( inputString[ testCharPosition ] == COMMAND_WORD_TERMINATOR ) {
      foundTheTerminator = true;
      terminatorPosition = testCharPosition;
   }
   else {
      testCharPosition = testCharPosition + 1;
   }
}
```

هاهنا المتحول الذي يحوي نتيجة البحث

إن كانت الشفرة جيدة كفاية، فهي تبدأ تُقرأ بالقرب من مستوى القصد، وتتجاوز شرح التعليقات للقصد من الشفرة. في هذه النقطة، قد يكون التعليق والشفرة مكرران بطريقة ما، لكنها مشكلة قلة من البرامج تعاني منها.

خطوة جيدة أخرى من أجل هذه الشفرة ستكون أن تُنشئ إجرائية تُدعى شيئاً ما مثل *FindCommandWordTerminator()* ونقل الشفرة من المثال إلى تلك الإجرائية.[1] تعليق يشرح تلك الفكرة مفيد لكنه مرجح أن يصبح غير دقيقاً أكثر من اسم الإجرائية عند تطور البرمجية.

ركز في تعليقات الفقرات على لمَ وليس كيف التعليقات التي تشرح كيف يُنفّذ شيء ما عادةً تعمل في مستوى لغة البرمجة وليس في مستوى المشكلة. تقريباً من المستحيل لتعليق يركز على كيفية تنفيذ عملية أن يشرح القصد من العملية، والتعليقات التي تخبر الكيفية هي غالباً مكررة. ماذا يخبرك التعليق التالي ولا تخبرك الشفرة ذاته؟



مثال جافا عن تعليق يركز على الكيفية

```
// if account flag is zero
if ( accountFlag == 0 ) ...
```

[1] **إشارة مرجعية** للمزيد عن نقل قسم من الشفرة إلى إجرائية خاصة، الق نظرة على "استخلاص إجرائية/استخلاص طريقة" في القسم 24. 3

لا يخبرك التعليق أي شيء زيادة على ما تقوم به الشفرة نفسها. ماذا عن هذا التعليق؟

مثال جافا عن تعليق يركز على *السبب*
```
// if establishing a new account
if ( accountFlag == 0 ) ...
```

هذا التعليق أفضل بكثير لأنه يخبرك شيئاً ما لا يمكن أن تعلمه من الشفرة نفسها. لا تزال الشفرة قابلة للتحسين عن طريق استخدام اسم نمط تعدادي ذي معنى بدلاً من 0 واسم متحول أفضل. إليك النسخة الأفضل من هذا التعليق والشفرة:

مثال جافا عن استخدام أسلوب جيد بالإضافة إلى تعليق عن "السبب"
```
// if establishing a new account
if ( accountType == AccountType.NewAccount ) ...
```

عندما تحصل الشفرة على هذه الدرجة من قابلية القراءة، من الجيد أن نسأل عن قيمة التعليق. في هذه الحالة، جُعل التعليق زائداً بتحسين الشفرة، وقد تنبغي إزالته. بطريقة بديلة، يمكن أن يزاح غرض التعليق بدقة، كهذا:

مثال جافا عن استخدام تعليق "عنوان قسم"
```
// establish a new account
if ( accountType == AccountType.NewAccount ) {
   ...
}
```

إن كان هذا التعليق يوثق كامل كتلة الشفرة اللاحقة لاختبار *if*، إنه يخدم كتعليق في مستوى التلخيص والاحتفاظ به مناسب كعنوان قسم للفقرة من الشفرة التي يشير إليها.

استخدم التعليقات لتهيئ القارئ لما هو آتٍ تخبر التعليقات الجيدة الشخص الذي يقرأ الشفرة ما هو متوقع. ينبغي أن يكون القارئ قادراً على مسح التعليقات فقط بنظرة والحصول على فكرة جيدة عمّ تقوم به الشفرة وعن أين يبحث عن نشاط محدد. نتيجة مباشرة لهذه القاعدة أن التعليق ينبغي دائماً أن يسبق الشفرة التي يصفها. هذه الفكرة لا تُعلّم دائماً في صفوف البرمجة، لكنها عُرف معترف به بشكل جيد في التطبيق التجاري.

اجعل كل تعليق مقبولاً رسمياً لا توجد أي فضيلة في التعليقات المتجاوزة للحد-الكثير جداً من التعليقات تُبهِم الشفرة التي تقصد أن توضحها. بدلاً من كتابة تعليقات إضافية، ضع الجهد الزائد في جعل الشفرة نفسها مقروءة أكثر.

وثق المفاجئات إن وجدت شيئاً ما ليس واضحاً من الشفرة نفسها، ضعه في تعليقات. إن استخدمت تقنية مخادعة بدلاً من واحدة مستقيمة لتحسّن الأداء، استخدم التعليقات لتشير إلى ما يمكن أن تكون التقنية المستقيمة لتحصي كمية الكسب في الأداء المنجز باستخدام الطريقة المخادعة. هاهنا مثال:

مثال سي ++ عن توثيق مفاجأة

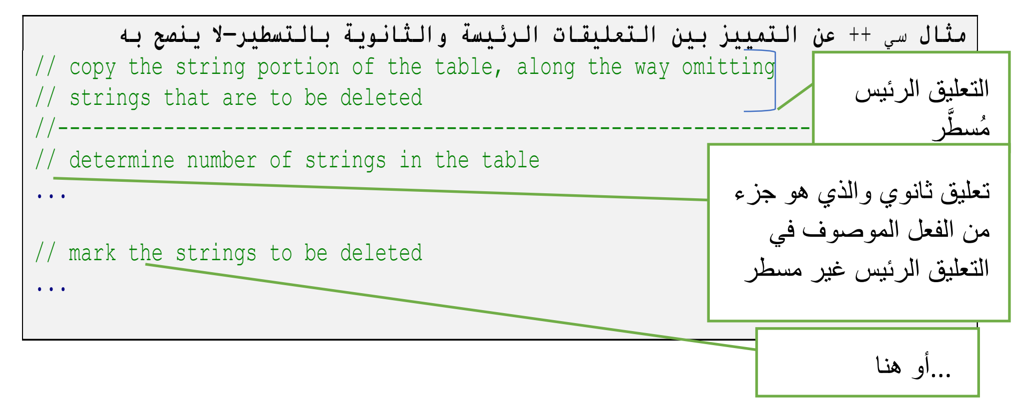
```
for ( element = 0; element < elementCount; element++ ) {
   // Use right shift to divide by two. Substituting the
   // right-shift operation cuts the loop time by 75%.
   elementList[ element ] = elementList[ element ] >> 1;
}
```

اختيار الإزاحة اليمنى في هذا المثال هو مثال عالمي. بين المبرمجين الخبراء، توجد معلومة شائعة بالنسبة للأعداد الصحيحة، هي أن الإزاحة اليمنى هي وظيفياً مكافئة للقسمة على 2.

إن كانت معلومة شائعة، لمَ توثقها؟ لأن الغرض من العملية ليس أن تقوم بإزاحة يمنى؛ إنه أن تنجز قسمة على 2. حقيقة أن الشفرة لم تستخدم التقنية الأكثر مناسبة لغرضها ملحوظة. أكثر بعد، معظم المترجمات تحسّن القسمة الصحيحة على 2 لتصبح إزاحة يمنى على كل حال، ما يعني أن التوضيح المُخفَّض ليس ضرورياً عادة. في هذه الحالة الخاصة، بوضوح، لم يحسّن المترجم القسمة على 2، والوقت الموفَّر سيكون ملحوظاً. مع التوثيق، سيرى المبرمج الذي يقرأ الشفرة الدافع من استخدام التقنية غير الواضحة. بدون التعليق، نفس المبرمج سيكون منجذباً إلى الهمهمة بأنّ الشفرة "ذكية" بلا ضرورة وبدون أي كسب ذي معنى في الأداء. عادةً هكذا همهمة مُسوَّغة، لذا فمن المهم أن توثق الاستثناء.

تجنب الاختصارات ينبغي أن تكون التعليقات مفهومة، ومقروءة بدون أي عمل لاكتشاف الاختصارات. تجنب الكل باستثناء الاختصارات الأكثر شيوعاً في التعليقات. مالم تكن تستخدم تعليقات نهاية السطر، استخدام الاختصارات ليس مغرياً عادةً. إن كنت تستخدم وكان الاستخدام مغرياً، تحقق أن الاختصارات هي ضربة أخرى للتقنية التي ضَربت برمجيات عدة فيما مضى.

ميّز بين التعليقات الرئيسة والتعليقات الثانوية في حالات قليلة، قد تريد التمييز بين المستويات المختلفة للتعليقات، مشيراً إلى أن التعليقات التفصيلية هي جزء من تعليق أعم سابق. تستطيع أن تتعامل مع هذا بزوج من الطرق. تستطيع أن تجرب تسطير التعليق الرئيس وعدم تسطير التعليق الثانوي:


مثال سي ++ عن التمييز بين التعليقات الرئيسة والثانوية بالتسطير-لا ينصح به

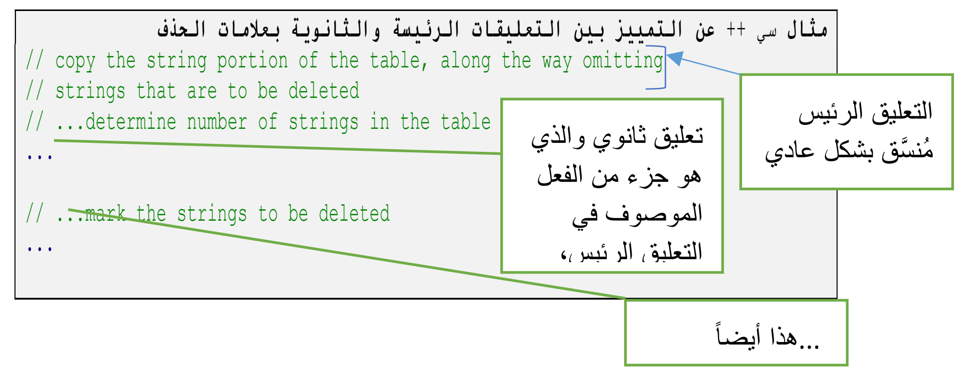
```
// copy the string portion of the table, along the way omitting
// strings that are to be deleted
//-----------------------------------------------------------------
// determine number of strings in the table
...

// mark the strings to be deleted
...
```

ضعف هذا النهج أنه يجبرك على تسطير تعليقات أكثر مما تحب فعلياً أن تسطر. إن سطّرت تعليقاً، فمن المفترض أن تكون كل التعليقات غير المسطّرة التي تليه تابعة له. كنتيجة، عندما تكتب التعليق الأول الذي ليس تابعاً للتعليق المسطّر، يتحتم جداً أن تسطّره وتبدأ الدورة من جديد. النتيجة تسطير زائد جداً عن الحد أو تسطير في بعض المناطق وعدم تسطير في مناطق أخرى بطريقة غير متماسكة.

لهذا النهج أشكال متعددة تمتلك كلها نفس المشكلة. إن وضعت التعليق الرئيس بحروف كبيرة كلها والتعليقات الثانوية بحروف صغيرة، ستُبدِّل المشكلة من الكثير جداً من تعليقات مسطّرة إلى الكثير جداً من تعليقات بحروف كبيرة. يستخدم بعض المبرمجين حروف كبيرة ابتدائية في التعليقات الرئيسة ولا حروف كبيرة ابتدائية في الثانوية، لكن هذا تلميح مرئي دقيق يمكن أن يُغفل ببساطة شديدة.

نهج أفضل أن تستخدم علامات الحذف في بداية التعليقات الثانوية:



نهج آخر، غالباً هو الأفضل، أن تضع عمليات التعليق الرئيس في إجرائية خاصة. ينبغي أن تكون الإجرائية منطقياً "مُسَطَّحة،" مع كل نشاطاتها إلى نفس المستوى المنطقي تقريباً. إن ميّزت الشفرة بين النشاطات الرئيسة والثانوية ضمن إجرائية، تكون الإجرائية غير مسطحة. وضع المجموعات المعقدة من النشاطات في إجرائيات خاصة يذهب باتجاه إجرائيتين مسطحتين منطقياً بدلاً من واحدة وعرة منطقياً.

هذا النقاش عن التعليقات الرئيسة والثانوية لا يُطبق على الشفرة المُعدّة للحلقات والشرطيات. في هكذا حالات، سيكون لديك غالباً تعليق واسع في أعلى الحلقة وتعليقات أكثر تفصيلاً عن العمليات ضمن الشفرة المُعدّة. في هذه الحالات، تؤمن المحاذاة فكرة عن التنظيم المنطقي للتعليقات. هذا النقاش يُطبّق فقط للفقرات التتابعية من الشفرة، والتي بها تُكوِّن عدة فقرات عملية كاملة وتكون بعض الفقرات تابعة لبعض آخر.

اكتب تعليقاً عن أي شيء حلَّ خطأً أو عن ميزة غير موثقة في اللغة أو البيئة إن كان خطأ، فقد يكون لم يُوثَّق بعد. حتى وإن وُثِّق في مكان ما، فلن يتسبب توثيقه مجدداً في شفرتك بالأذى. إن كانت ميزة غير موثقة، فحسب التعريف إنها غير موثقة في أي مكان آخر وينبغي أن توثق في شفرتك.

هب أنك وجدت إجرائية مكتبة *WriteData (data, numItems, blockSize)* تعمل بشكل صحيح ما عدا حالة مساواة *blockSize* لـ 500. إنها تعمل بشكل جيد من أجل 499 و 501 وكل القيم الأخرى التي

جربتها في حياتك، لكنك وجدت أنها تحتوي خلل يظهر فقط عندما يساوي 500 *blockSize*. في الشفرة التي تستخدم *WriteData()*، وثق سبب إنشائك لحالة خاصة عندما يكون 500 *blockSize*. هاهنا مثال على كيف يمكن أن يبدو:

مثال جافا عن توثيق حل لخطأ

```
blockSize = optimalBlockSize( numItems, sizePerItem );

/* The following code is necessary to work around an error in
WriteData() that appears only when the third parameter
equals 500. '500' has been replaced with a named constant
for clarity.
*/
if ( blockSize == WRITEDATA_BROKEN_SIZE ) {
   blockSize = WRITEDATA_WORKAROUND_SIZE;
}
WriteData ( file, data, blockSize );
```

برر خروقات أسلوب البرمجة الجيد إن وجب عليك أن تخرق أسلوب برمجة جيد، اشرح لمَ. هذا سيمنع مبرمجاً حسن النية من تغيير الشفرة إلى أسلوب أفضل، ربما محطماً شفرتك. سيجعل الشرح معرفة ما كنت تقوم به وأنك لست مغفّلاً أمراً واضحاً-أعطِ نفسك مصداقية عندما تكون المصداقية مبارزة بين اثنين!

لا تعلق على الشفرة المخادعة؛ أعد كتابتها هاك تعليقاً من مشروع عملت عليه:

شفرة مرعبة

مثال سي ++ عن التعليق على شفرة ذكية

```
// VERY IMPORTANT NOTE:
// The constructor for this class takes a reference to a UiPublication.
// The UiPublication object MUST NOT BE DESTROYED before the
// DatabasePublication object. If it is, the DatabasePublication object
// will cause the program to die a horrible death.
```

هذا مثال جيد عن واحدة من الأجزاء الأكثر شيوعاً ومجازفةً في الفلكلور البرمجي؛ ينبغي أن تُستخدم التعليقات للتوثيق الأقسام "المخادعة" أو "الحساسة" من الشفرة بشكل خاص. التفسير هو أنه ينبغي أن يعلم الناس أنهم بحاجة الحذر عندما يعملون في مناطق محددة.

هذه فكرة مرعبة.

التعليق على الشفرة المخادعة هو بالضبط النهج الخاطئ في الاتباع. لا تستطيع التعليقات أن تنقذ الشفرة المسببة للمصاعب. كما أكد كيرنيغان وبلاوغير، "لا توثق الشفرة السيئة-أعد كتابتها" (1978).

بيانات مثبتة

وجدت دراسة واحدة أن مناطق الشفرة المصدرية التي تحوي عدداً ضخماً من التعليقات كانت تميل أيضاً لتمتلك معظم العيوب وتستهلك معظم جهود التطوير (لند وفيارافان 1989). افترض الكاتبان أن المبرمجين يميلون للتعليق بكثافة على الشفرة الصعبة.

عندما يقول أحد ما، "هذه حقاً شفرة مخادعة،" أسمعهم يقولون، "هذه حقاً شفرة *سيئة*." إن بدا شيء ما مخادعاً لك، سيكون غير مفهوم لشخص غيرك. حتى الأشياء التي لا تبدو بشكل كامل مخادعة لك، يمكن أن تبدو معقودة بشكل مستحيل لشخص آخر لم يرى هذه الخدعة من قبل. إن وجب عليك أن تسأل نفسك "هل هذا مخادع؟" فهو كذلك. تستطيع دائماً ان تجد كتابة ثانية ليست مخادعة، لذا أعد كتابة الشفرة. اجعل شفرتك جيدة جداً حتى لا تحتاج تعليقات، ومن ثم علق عليها لتجعلها أفضل أكثر.

تُطبق هذه النصيحة بشكل رئيس على الشفرة التي تكتبها للمرة الأولى. إن كنت تصون برنامجاً ولا تمتلك الحرية في إعادة كتابة الشفرة السيئة، تكون كتابة التعليقات على الأجزاء المخادعة تطبيقاً جيداً.

كتابة تعليقات على التصريحات عن البيانات

تصف التعليقات على التصريحات عن المتحولات جوانب من المتحول لا يستطيع اسم المتحول أن يصفها.[1] من المهم أن توثق البيانات بحذر، شركة واحدة على الأقل والتي درست تطبيقاتها الخاصة خَلُصَث إلى أن التعليقات على البيانات كانت أكثر أهمية من التعليقات على العمليات التي تُستخدم فيها البيانات (إس دي سي، في غلاس 1982). هاهنا بعض التوجيهات من أجل التعليق على البيانات:

اكتب واحدات البيانات الرقمية في تعليقات إن كان رقم يمثل طولاً، حدد إن كان الطول بالإنشات، أو الأقدام، أو الكيلومترات. إن كان زمناً، حدد إن كان بالثواني الماضية منذ 1-1-1980، أو بميليات الثانية منذ بداية البرنامج، وهكذا. إن كان إحداثيات، حدد إن كان يمثل خط الطول أو العرض أو الارتفاع، وإن كان بالراديان أو الدرجات، أو إن كان بنظام إحداثيات *X, Y, Z* مركزه مركز الأرض؛ وهلم جراً. لا تفترض أن الواحدات ظاهرة. بالنسبة للمبرمجين الجدد، لن تكون كذلك. لشخص آخر يعمل على قسم آخر من النظام، لن تكون كذلك. بعد أن يُعدّل البرنامج بشكل جوهري، لن تكون كذلك.

كبدل، في العديد من الحالات ينبغي أن تُضمّن الواحدات في أسماء المتحولات بدلاً من التعليقات. تعبير مثل `distanceToSurface = marsLanderAltitude` يبدو وكأنه صحيح، لكن `distanceToSurfaceInMeters = marsLanderAltitudeInFeet` يكشف عن خطأ بيّن.

[1] **إشارة مرجعية** لتفاصيل عن تنسيق البيانات، الق نظرة على "رسم التصريحات عن البيانات" في القسم 32. 5. لتفاصيل عن كيفية استخدام البيانات بفعالية، الق نظرة على الفصول من 10 إلى 13.

علق على مجال القيم الرقمية المسموحة [1] إن كان متحول يمتلك مجالاً متوقعاً من القيم، وثق هذا المجال المتوقع. أحد الميزات القوية للغة البرمجة **آدا**، كانت قدرتها على تقييد القيم المسموحة لمتحول رقمي إلى مجال من القيم. إن كانت لغتك لا تدعم هذه المقدرة (ومعظم اللغات لا تدعم)، استخدم تعليقاً لتوثق المجال المتوقع للقيم. مثلاً، إن كان المتحول يمثل كمية المال بالدولارات، حدد أنه ينبغي أن يكون بين 105v و 125v.

دوّن المعاني المرمزة في تعليقات إن كانت لغتك تدعم الأنماط التعدادية-مثل **سي ++ وفيجوال بيسك**-استخدمها لتعبّر عن المعاني المرمزة. إن لم تكن، استخدم التعليقات لتحدد ما تمثله كل قيمة-واستخدم ثابتاً مسمى بدلاً من الحرفية لكل واحدة من القيم. إن مثّل متحول أنواع التيارات الكهربائية، اكتب حقيقة أن 1 تمثل التيار المتناوب، و2 تمثل التيار المستمر و3 تمثل غير محدد في تعليق.

هاهنا مثال عن توثيق التصريحات عن المتحولات يوضح التوصيات الثلاثة السابقة-كل معلومات المجال معطاة في تعليقات:

مثال فيجوال بيسك **عن تصريحات عن بيانات معلق عليها بشكل جميل**

```
Dim cursorX As Integer ' horizontal cursor position; ranges from 1..MaxCols
Dim cursorY As Integer ' vertical cursor position; ranges from 1..MaxRows

Dim antennaLength As Long ' length of antenna in meters; range is >= 2
Dim signalStrength As Integer ' strength of signal in kilowatts; range is >= 1

Dim characterCode As Integer ' ASCII character code; ranges from 0..255
Dim characterAttribute As Integer ' 0=Plain; 1=Italic; 2=Bold; 3=BoldItalic
Dim characterSize As Integer ' size of character in points; ranges from 4..127
```

اكتب القيود على بيانات الدخل في تعليقات قد تأتي بيانات الدخل من وسيط دخل أو ملف أو إدخال مباشر من المستخدم. يُطبّق التوجيه السابق على وسطاء دخل إجرائية بمقدار ما يطبق على أنواع أخرى من البيانات. تأكد أن القيم المتوقعة وغير المتوقعة مُوثَّقة. التعليقات هي إحدى الطرق لتوثيق أن ليس من المفترض أبداً لإجرائية أن تستقبل بيانات محددة. التأكيدات هي طريقة أخرى لتوثيق المجالات الصالحة، وإن استخدمتها ستصبح الشفرة أقرب بكثير إلى تلك الشفرة التي تختبر نفسها ذاتياً.

وثّق الرايات إلى مستوى البت [2] إن استخدم متحول كحقل بت واحد، وثّق معنى كل بت:

[1] **إشارة مرجعية** استخدام التأكيدات في بداية ونهاية إجرائية لتتأكد أن قيم المتحولات تقع ضمن المجال المحدد، هو تقنية أقوى لتوثيق المجالات المسموحة للمتحولات. للمزيد من التفاصيل، راجع القسم 8 .2، "التأكيدات."

[2] **إشارة مرجعية** لتفاصيل عن تسمية متحولات الرايات، راجع "تسمية متحولات الحالة" في القسم 11 .2

```
مثال فيجوال بيسك عن توثيق الرايات على مستوى البت
' The meanings of the bits in statusFlags are as follows, from most
' significant bit to least significant bit:
' MSB       0     error detected: 1=yes, 0=no
'           1-2   kind of error: 0=syntax, 1=warning, 2=severe, 3=fatal
'           3     reserved (should be 0)
'           4     printer status: 1=ready, 0=not ready
'           ...
'           14    not used (should be 0)
' LSB       15-32 not used (should be 0)
Dim statusFlags As Integer
```

إن كان المثال مكتوباً في **سي ++**، فإنه سيستدعي قواعد كتابة "حقل البت" عندها ستكون معاني حقل البت موثقة ذاتياً.

الصق التعليقات المتعلقة بمتحول باسم هذا المتحول إن كان لديك تعليق يشير إلى متحول محدد، تأكد أن التعليق يُحدَّث متى يُحدَّث المتحول. إحدى الطرق لتحسّن الشذوذات عن التعديلات المتماسكة هي أن تلصق التعليق باسم المتحول. بهذه الطريقة، بحث السلسلة النصية عن اسم المتحول سيجد التعليق بالإضافة إلى المتحول.

وثّق البيانات الشاملة[1] إن استخدمت البيانات الشاملة، علق على كل واحدة بشكل جيد في النقطة التي تم التصريح عنها بها. ينبغي أن يحدد التعليق الغرض من البيانات وسبب حاجتها لتكون شاملة. في كل نقطة تُستخدم بها البيانات الشاملة، اجعل كون البيانات شاملة أمراً واضحاً. عرف التسمية هو الخيار الأول للإضاءة على حالة المتحول الشاملة. إن لم يُستخدم عرف تسمية، يمكن للتعليقات أن تسد الثغرة.

كتابة التعليقات على بنى التحكم

الفراغ قبل بنية التحكم هو عادةً مكان طبيعي لوضع تعليق.[2] إن كانت عبارة `if` أو *case*، بإمكانك أن تقدّم السبب عن القرار وملخصاً عن النتيجة. إن كانت حلقة، تستطيع أن تحدد الغرض من الحلقة.

[1] **إشارة مرجعية** لتفاصيل عن استخدام المتحولات الشاملة، انظر القسم 13. 3 "البيانات الشاملة."

[2] **إشارة مرجعية** لتفاصيل أخرى عن بنى التحكم، راجع القسم 31. 3، "أساليب التنسيق،" والقسم 31. 4 "رسم بنى التحكم،" والفصول من 14 إلى 19.

مثال سي ++ عن التعليق على الغرض من بنية تحكم

```
// copy input field up to comma
while ( ( *inputString != ',' ) && ( *inputString != END_OF_STRING ) ) {
   *field = *inputString;
   field++;
   inputString++;
} // while -- copy input field
*field = END_OF_STRING;
if ( *inputString != END_OF_STRING ) {
   // read past comma and subsequent blanks to get to the next input field
   inputString++;
   while ( ( *inputString == ' ' ) && ( *inputString != END_OF_STRING ) ) {
      inputString++;
   }
} // if -- at end of string
```

الغرض من الحلقة

نهاية الحلقة (مفيدة للحلقات الأطول والمعششة-مع أن الحاجة إلى هكذا تعليق تشير إلى شفرة معقدة فوق الحد)

الغرض من الحلقة. موقع التعليق يجعل تهيئة *inputString* من أجل الحلقة أمراً واضحاً

يقترح هذا المثال بعض التوجيهات:

ضع تعليقاً قبل كل* `if` *و* `case` *وكتلة من العبارات هكذا مكان هو بقعة طبيعة للتعليق، وعادة ما تحتاج هذه البنى إلى

شرح. استخدم تعليقاً لتوضح الغرض من تركيبة التحكم.

ضع تعليقاً على نهاية كل تركيبة تحكم استخدم تعليقاً لتظهر ما هو الذي انتهي-مثلاً،

`} // for clientIndex — process record for each client`

تعليق في نهاية حلقة طويلة يساعد بشكل خاص على توضيح تعشيش الحلقات. هاهنا مثال جافا عن استخدام التعليقات لتوضيح نهايات بنى الحلقات:

مثال جافا عن استخدام التعليقات لتوضيح التعشيش

```
for ( tableIndex = 0; tableIndex < tableCount; tableIndex++ ) {
   while ( recordIndex < recordCount ) {
      if ( !IllegalRecordNumber( recordIndex ) ) {
         ...
      } // if
   } // while
} // for
```

هذه التعليقات تحدد أي بنية تحكم تنتهي

تقنية التعليق هذه تزود بدلالة مرئية عن البنيان المنطقي المعطى بمحاذاة الشفرة. لا تحتاج أن تستخدم هذه التقنية للحلقات القصيرة غير المعششة. عندما يكون التعشيش عميقاً أو تكون الحلقات طويلة، على كل، تدفع التقنية مستحقاتها.

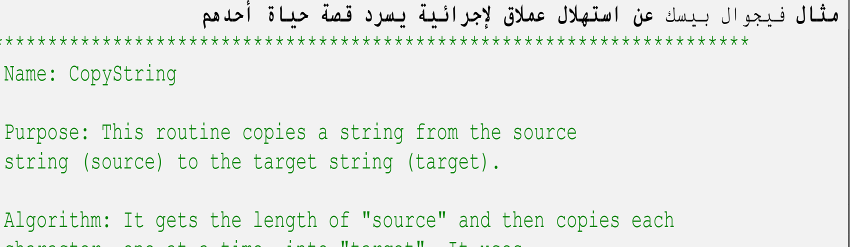
عامل تعليقات نهاية الحلقة كتحذيرات تشير إلى شفرة معقدة إن كانت حلقة معقدة كفاية حتى تحتاج إلى تعليق نهاية حلقة، عامل التعليق كإشارة تحذير: قد يلزم تبسيط الحلقة. نفس القاعدة تُطبّق على اختبارات *if* المعقدة وعبارات *case*.

تقدم تعليقات نهاية الحلقة أفكاراً مفيدة عن البنيان المنطقي، لكن كتابتها بداية ثم صيانتها يمكن أن تصبح مملة. الطريقة الأمثل لتجنب هكذا عمل ممل هو غالباً أن تعيد كتابة أي شفرة معقدة كفاية لدرجة أنها تتطلب توثيقاً مملاً.

كتابة التعليقات على الإجرائيات

التعليقات في مستوى الإجرائيات هي موضوع لبعض أسوأ النصائح في كتب علم الحاسوب النمطية.[1] تحثك كتب عديدة إلى تكديس كومة من المعلومات في أعلى كل إجرائية، بغض النظر عن حجمها أو تعقيدها:





مثال فيجوال بيسك **عن استهلال عملاق لإجرائية يسرد قصة حياة أحدهم**

```
'**********************************************************************
' Name: CopyString
'
' Purpose: This routine copies a string from the source
' string (source) to the target string (target).
'
' Algorithm: It gets the length of "source" and then copies each
' character, one at a time, into "target". It uses
' the loop index as an array index into both "source"
' and "target" and increments the loop/array index
' after each character is copied.
'
' Inputs: input The string to be copied
'
' Outputs: output The string to receive the copy of "input"
'
' Interface Assumptions: None
'
' Modification History: None
'
' Author: Dwight K. Coder
' Date Created: 10/1/04
' Phone: (555) 222-2255
' SSN: 111-22-3333
' Eye Color: Green
' Maiden Name: None
' Blood Type: AB-
' Mother's Maiden Name: None
' Favorite Car: Pontiac Aztek
' Personalized License Plate: "Tek-ie"
'**********************************************************************
```

[1] **إشارة مرجعية** لتفاصيل عن تنسيق الإجرائيات، راجع القسم 31. 7. لتفاصيل عن كيفية إنشاء إجرائيات عالية الجودة، راجع القسم 7.

هذه تفاهة. *CopyString* هي إجرائية بسيطة على نحو راجح-ربما أقل من خمسة أسطر من الشفرة. التعليق خارج كلياً عن مقدار استيعاب مقياس الإجرائية. الأجزاء عن ***الغرض من*** الإجرائية و***خوارزميتها*** مصطنعة لأنه من الصعب أن تشرح شيئاً ما ببساطة *CopyString* في مستوى من التفصيل بين "نسخ سلسلة" والشفرة نفسها. تعليقات غطاء السخان: ***فرضيات الواجهة وتاريخ التعديل*** ليست مفيدة أيضاً-إنها فقط تستهلك مساحة في التعداد. طلب اسم الكاتب مكرر مع معلومات يمكن الحصول عليها بدقة أكبر من نظام التحكم بالإصدار. أن تفرض كل هذه العوامل لكل إجرائية هو وصفة للتعليقات غير الدقيقة وفشل الصيانة. إنها أعمال مصطنعة كثيرة والتي أبداً لا تدفع مستحقاتها.

مشكلة أخرى مع رؤوس الإجرائيات الثقيلة هي أنها تثبط التركيز الجيد على الشفرة-تكون كلف إنشاء إجرائية جديدة عالية جداً بحيث سيميل المبرمجون خطأً إلى جانب إنشاء إجرائيات أقل، وليس أكثر. ينبغي أن تحثّ أعراف كتابة الشفرة على التطبيقات الجيدة؛ تقوم رؤوس الإجرائيات الثقيلة بالعكس.

إليك بعض التوجيهات للتعليق على الإجرائيات:

ابق التعليقات قريبة من الشفرة التي تصف أحد أسباب أن استهلال إجرائية لا ينبغي أن يحتوي توثيقاً ضخماً هو أن هكذا تطبيق يضع التعليقات بعيدة عن أجزاء الإجرائية التي تصفها. خلال الصيانة، التعليقات التي تكون بعيدة عن الشفرة تميل لألا تُصان مع الشفرة. وتبدأ الشفرة والتعليقات بالاختلاف، وفجأة تصبح التعليقات بلا قيمة. بدلاً، اتّبع **مبدأ التقريب** وضَعْ التعليقات أقرب ما يمكن إلى الشفرة التي تصفها. هناك ستمتلك فرصة أكبر لتُصان، وستواصل كونها ذات قيمة.

وُصفت عدة مكونات من استهلال الإجرائية في الأسفل وينبغي أن تُضمَّن حسب الحاجة. من أجل راحتك، أنشئ استهلال توثيق غطاء السخان. لكن فقط لا تشعر أنك ملزم بتضمين كل المعلومات في كل الحالات. املأ الأجزاء التي تهم، واحذف الباقي.

صف كل إجرائية بجملة أو اثنتين في أعلى الإجرائية[1] إن لم تستطع وصف الإجرائية بجملة قصيرة أو اثنتين. قد تحتاج أن تفكر بجد أكثر بخصوص ما المفترض أن تقوم به. الصعوبة في إنشاء وصف قصير هي إشارة على أن التصميم ليس بمقدار الجودة التي ينبغي أن تكون. عد إلى لوح رسومات التصميم وحاول مجدداً. ينبغي أن تكون جملة التلخيص القصيرة حاضرة في كل الإجرائيات عملياً ما عدا إجرائيات النفاذ البسيطة *Get* و*Set*.

[1] **إشارة مرجعية** أسماء الإجرائيات الجيدة هي مفتاح لتوثيق الإجرائيات. لتفاصيل حول كيفية إنشائها، راجع القسم 7. 3، "أسماء إجرائيات جيدة."

وثق الوسطاء في مكان التصريح عنها أسهل طريقة لتوثيق متحولات الدخل والخرج هي وضع تعليقات بجانب كل تصريح عن متحول.

```
مثال جافا عن توثيق بيانات الدخل والخرج في مكان التصريح عنها-تطبيق جيد
public void InsertionSort(
   int[] dataToSort, // elements to sort in locations firstElement..lastElement
   int firstElement, // index of first element to sort (>=0)
   int lastElement // index of last element to sort (<= MAX_ELEMENTS)
)
```

هذا التطبيق هو استثناء جيد لقاعدة عدم استخدام تعليقات نهاية السطر؛ إنها مفيدة بشكل استثنائي في توثيق وسطاء التدخل والخرج.[1] هذه الفرصة لكتابة التعليق هي توضيح جيد لقيمة استخدام المحاذاة المعيارية بدلاً من محاذاة سطر النهاية من أجل لائحة وسطاء إجرائية-لن تمتلك مساحة من أجل تعليقات نهاية سطر ذات معنى أن استخدمت محاذاة نهاية السطر. أرهقت التعليقات في المثال من أجل الحصول على المساحة حتى مع المحاذاة المعيارية. هذا المثال يشرح أيضاً أن التعليقات ليست الشكل الوحيد للتوثيق. إن كانت أسماء متحولاتك جيدة كفاية، قد تكون قادراً على تجاوز كتابة التعليقات عليها. أخيراً، الحاجة إلى توثيق متحولات الدخل والخرج هي سبب جيد لتجنب البيانات الشاملة. أين توثقها؟ بشكل متوقع، توثق البيانات الشاملة في الاستهلال العملاق. هذا يذهب باتجاه المزيد من العمل و، لسوء الحظ، في التطبيق، يعنى عادة أن البيانات الشاملة لم تُوثَّق. هذا سيء جداً لأن البيانات الشاملة تحتاج أن تُوثَّق على الأقل بمقدار أي شيء آخر.

استفد من ميزات أدوات توثيق الشفرة مثل Javadoc إن كانت الشفرة في المثال السابق مكتوبة فعلاً ب**جافا**، فإنك تمتلك القدرة الإضافية على تهيئة الشفرة لتستفيد من ميزات أداة استخراج التوثيق الخاصة ب**جافا**، Javadoc. في تلك الحالة، "توثيق الوسطاء في مكان التصريح عنها" سيتغير ليبدو كالتالي:

```
مثال جافا عن توثيق بيانات الدخل والخرج للاستفادة من ميزات Javadoc
/**
 * ... <description of the routine> ...
 *
 * @param dataToSort elements to sort in locations firstElement..lastElement
 * @param firstElement index of first element to sort (>=0)
 * @param lastElement index of last element to sort (<= MAX_ELEMENTS)
 */
public void InsertionSort(
   int[] dataToSort,
   int firstElement,
   int lastElement
)
```

مع أداة مثل Javadoc، فائدة تهيئة الشفرة لاستخلاص التوثيق تفوق المخاطر المرتبطة بفصل وصف المتحول عن تصريح المتحول. إن لم تكن تعمل في بيئة تدعم استخلاص التوثيق، مثل Javadoc، فعادة، سيكون من

[1] **إشارة مرجعية** نوقشت تعليقات نهاية السطر بتفصيل أكبر في "تعليقات نهاية السطر ومشاكلها" سابقاً في هذا القسم.

الأفضل لك أن تحافظ على التعليقات أقرب إلى أسماء المتحولات لتتجنب التحريرات غير المتماسكة وتكرار الأسماء نفسها.

ميّز بين بيانات الدخل والخرج من المفيد معرفة أي البيانات تُستخدم كدخل وأيها تُستخدم كخرج، تجعل **فيجوال بيسك** هذا سهلاً نسبياً لأن بيانات الخرج مسبوقة بالكلمة المفتاحية *ByRef* وبيانات الدخل مسبوقة بالكلمة المفتاحية *ByVal*. إن كانت لغتك لا تدعم هكذا تمييز بشكل آلي، ضعه في تعليق. هاهنا مثال في **سي** ++:[1]

```
مثال سي ++ عن التمييز بين بيانات الدخل والخرج
void StringCopy(
   char *target,        // out: string to copy to
   const char *source   // in: string to copy from
)
...
```

تكون تصريحات الإجرائيات في لغة **سي** ++ مخادعة قليلاً لأن النجمة (*) في بعض الأحيان تحدد أن الوسيط هو وسيط خرج، وفي الكثير من الأحيان تعني فقط أن التعامل مع المتحول كمؤشر أسهل من نمط غير المؤشر. فعادة، من الأفضل لك تعريف وسطاء الدخل والخرج بصراحة.

إن كانت إجرائياتك قصيرة كفاية وكنت تحافظ على تمييز واضح بين بيانات الدخل والخرج، فقد يكون توثيق حالة الدخل أو الخرج للبيانات غير ضروري. إن كانت الإجرائية أطول، على كل، فسيكون التوثيق خدمة مفيدة لأي شخص يقرأ الإجرائية.

وثّق افتراضات الواجهة[2] يمكن النظر إلى توثيق افتراضات الواجهة كفئة فرعية من توصيات كتابة التعليقات الأخرى. إن قمت بأي افتراض حول حالة المتحولات التي تستقبلها-القيم القانونية وغير القانونية، وكون المصفوفات مرتبة، وكون البيانات الأعضاء مهيئة أو تحتوي على قيم جيدة فقط، وهكذا-وثقه إما في استهلال الإجرائية أو حيث صُرِّح عنها. ينبغي أن يكون هذا التوثيق موجوداً في كل الإجرائيات عملياً.

تأكد أن البيانات الشاملة التي استخدمت موثقة. المتحول الشامل هو واجهة للإجرائية بمقدار أي شيء آخر وهو حتى الأكثر مجازفة لأنه بطريقة ما لا يبدو كشيء مجازف.

وأنت تكتب الإجرائية عندما تدرك أنك تقوم بافتراضات الواجهة، اكتبها مباشرة.

[1] **إشارة مرجعية** ترتيب هذه الوسطاء يتبع الترتيب المعياري لإجرائيات **سي** ++ لكنه يتعارض مع التطبيقات الأكثر عمومية. لتفاصيل، راجع "ضع الوسطاء في ترتيب دخل-تعديل-خرج" في القسم 7. 5. لتفاصيل عن استخدام أعراف التسمية للتمييز بين بيانات الدخل والخرج، راجع القسم 11. 4.

[2] **إشارة مرجعية** لتفاصيل عن الاعتبارات الأخرى حول واجهات الإجرائيات، راجع القسم 7. 5، "كيف تستخدم وسطاء إجرائية." لتوثق الافتراضات باستخدام التأكيدات، راجع "استخدم التأكيدات لتوثق وتؤكد الشروط المسبقة والشروط اللاحقة" في القسم 8. 2.

علّق على قيود الإجرائية إن كانت الإجرائية تقدم نتيجة رقمية، حدد دقة النتيجة. إن كانت الحسابات غير معرفة تحت بعض الشروط، وثق هذه الشروط. إن امتلكت الإجرائية سلوكاً افتراضياً عندما تدخل في مشكلة. وثّق هذا السلوك. إن كان من المتوقع أن تعمل الإجرائية فقط على مصفوفات أو جداول بحجوم محددة، حدد ذلك. إن كنت تعرف شيئاً عن تعديلات على البرنامج ستعطل الإجرائية، وثقها. إن اصطدمت ب "كلمات تقال لكن كتابتها غير قانونية" gotchas خلال تطوير الإجرائية، وثقها أيضاً.

وثق آثار الإجرائية الشاملة إن كانت الإجرائية تعدل بيانات شاملة، صف بالضبط ما تفعل مع البيانات الشاملة. كما ذُكر في القسم 13. 3، "البيانات الشاملة،" تعديل البيانات الشاملة هو طبقة حجم أكثر خطورة من مجرد قراءتها، لذا ينبغي أن تنجز التعديلات بحذر، كتابة توثيق واضح هو جزء من الحذر. كما هي العادة، إن أصبحت كتابة التوثيق أمراً متعباً جداً، أعد كتابة الشفرة لتخفّض البيانات الشاملة.

وثق مصدر الخوارزميات التي استُخدمت إن استخدمت خوارزمية من كتاب أو مجلة، وثق الإصدار ورقم الصفحة التي أخذتها منها. إن طورت الخوارزمية بنفسك، حدد مكاناً يستطيع القارئ أن يجد فيه الملاحظات التي كتبتها عنها.

استخدم التعليقات لتعلّم أجزاء من برنامجك يستخدم بعض المبرمجين التعليقات ليعلّموا أجزاء من برامجهم بحيث يستطيعون أن يجدوها بسهولة. واحدة من هكذا تقنيات في **سي ++ وجافا** أن تعلّم أعلى كل إجرائية بتعليق يبدأ بهذه المحارف:

/**

يسمح لك هذا بالقفز من إجرائية إلى إجرائية عن طريق القيام ببحث عن **/ أو أن تستخدم محررك لتقفز بشكل آلي إن كان يدعم ذلك.

أن تعلّم الأنواع المختلفة من التعليقات بأشكال مختلفة هي تقنية مشابهة، بالاعتماد على ما تصف. مثلاً، في **سي ++** تستطيع استخدام *@keyword*، حيث تكون *keyword* هي شفرة تستخدمها لتحدد نوع التعليق. التعليق *@param* يمكن أن يحدد أن التعليق يصف وسيطاً لإجرائية، و@version يمكن أن يحدد معلومات إصدار الملف، و@throws يمكن أن توثق الاستثناءات التي ترميها إجرائية، وهكذا. تسمح لك هذه التقنيات باستخدام أدوات لاستخلاص أنواع مختلفة من المعلومات عن ملفاتك المصدرية. مثلاً، يمكن البحث عن *@throws*لتحصل على توثيق عن كل الاستثناءات التي ترميها كل الإجرائيات في برنامج.

هذا العرف الخاص ب**سي ++** يعتمد على عرف Javadoc،[1] الذي هو عرف توثيق واجهات تم تأسيسه بشكل جيد من أجل برامج **جافا** (*java.sun.com/j2se/Javadoc/*). تستطيع أن تعرف أعرافك الخاصة في لغات أخرى.

كتابة التعليقات على الصفوف والملفات والبرامج

الصفوف والملفات والبرامج كلها مميزة بحقيقة أنها تحتوي على عدة إجرائيات.[2] ينبغي أن يحتوي الملف أو الصف على مجموعة من الإجرائيات المترابطة. يحتوي البرنامج على كل الإجرائيات في البرنامج. مهمة التوثيق في كل حالة هي أن تؤمن نظرة عالية المستوى ذات معنى لمحتوى الملف أو الصف أو البرنامج.

توجيهات عامة لتوثيق الصف

من أجل كل صف، استخدم تعليق كتلة لتصف الخصائص العامة للصف:

صِف نهج تصميم الصف تكون تعليقات النظرة العامة، التي تؤمن معلومات لا يمكن الحصول عليها حالاً بتطبيق الهندسة العكسية على تفاصيل كتابة الشفرة، مفيدة بشكل خاص. صِف فلسفة تصميم الصف، ونهج التصميم الكلي، وبدائل التصميم المأخوذة بعين الاعتبار والمهملة، وهلم جراً.

صِف القيود وافتراضات الاستخدام وهلم جراً بشكل مشابه للإجرائيات، تأكد أن تصف أي قيد مفروض من قبل تصميم الصف. أيضاً صِف الافتراضات عن بيانات الدخل والخرج، ومسؤوليات التعامل مع الأخطاء، والآثار الشاملة، ومصادر الخوارزميات، وهلم جراً.

علق على واجهة الصف أيستطيع مبرمج آخر فهم كيفية استخدام صف بدون النظر إلى تحقيق الصف؟ إن كان لا، فتغليف الصف في خطر حقيقي. ينبغي أن تحتوي واجهة الصف على كل المعلومات التي يحتاجها أي شخص ليستخدم الصف. عُرف Javadoc هو فرض، على الأقل، توثيق لكل وسيط ولكل قيمة معادة (**سن ميكروسيستمز** 2000). ينبغي أن يُنجز هذا الأمر من أجل كل الإجرائيات المكشوفة في كل صف (بلوتش 2001).

لا توثق تفاصيل التحقيق في واجهة الصف قاعدة رئيسة في التغليف هي أن تكشف فقط المعلومات حسب قاعدة الحاجة إلى المعرفة: إن كان لديك أي استفسار عن حاجة المعلومات للكشف أو لا، فالخيار

[1] cc2e.com/3259

[2] **إشارة مرجعية** لتفاصيل عن التنسيق، راجع القسم 31. *، "رسم الصفوف." لتفاصيل عن استخدام الصفوف، راجع الفصل 6، "صفوف ناجحة."

الافتراضي هو أن تبقيها مخفية. بناء على ذلك، ينبغي أن تحتوي ملفات واجهة الصف على المعلومات اللازمة لاستخدام الصف وليس المعلومات اللازمة للتحقيق أو صيانة الأشغال الداخلية للصف.

توجيهات عامة لتوثيق الملف

في أعلى الملف، استخدم تعليق كتلة لتصف محتوى الملف:

صِف الغرض من ومحتويات كل ملف ينبغي أن يصف تعليق رأس الملف الصفوف والإجرائيات المُضمَّنة في الملف. إن كانت كل إجرائيات البرنامج في ملف واحد، فغرض الملف واضح تماماً-هو الملف الذي يحتوي كامل البرنامج. إن كان الغرض من الملف أن يحتوي على صف خاص واحد، فالغرض أيضاً واضح-إنه الملف الذي يحتوي صفاً بنفس الاسم.

إن كان الملف يحتوي على أكثر من صف واحد، اشرح لمَ تحتاج الصفوف أن تُجمَّع في ملف مفرد.

إن تمت القسمة إلى ملفات مصدرية متعددة لسبب ما غير الوحدوية، سيكون وصف جيد للملف مساعداً أكثر للمبرمج الذي يعدل البرنامج. إن كان أحد ما يبحث عن إجرائية تقوم بالأمر س، هل يحتوي رأس الملف على تعليق يساعد ذلك الشخص على تحديد إن كان هذا الملف يحتوي على هكذا إجرائية؟

ضع اسمك وبريدك الالكتروني ورقم هاتفك في تعليق الكتلة يصبح التأليف والمسؤولية الرئيسة على مناطق محددة من الشفرة أمراً مهماً في المشاريع الكبيرة. تتمكن المشاريع الصغيرة (أقل من 10 أشخاص) من استخدام نُهج التطوير التعاوني، كالملكية المشتركة للشفرة والتي فيها كل عناصر الفريق مسؤولون بتساو عن كل أقسام الشفرة. تفرض أنظمة أكبر على المبرمجين التخصص في مناطق من الشفرة، ما يجعل الملكية المشتركة للشفرة لكامل الفريق أمراً غير قابل للتطبيق.

في تلك الحالة، التأليف هو معلومة مهمة ينبغي أن تكون في التعداد المصدري. إنها تعطي المبرمجين الآخرين الذين يعملون على الشفرة فكرة عن أسلوب البرمجة، وتعرّفهم شخصاً ما ليتصلوا به إن احتاجوا مساعدة. حسب المكان الذي تعمل عليه إن كان إجرائيات مفردة، أو صفوفا، أو برامجا ينبغي أن تُضمن معلومات الكاتب في مستوى الإجرائية، أو الصف، أو البرنامج.

ضمن علامة للتحكم بالإصدار سيحشر العديد من أدوات التحكم بالإصدار معلومات الإصدار في ملف. في **سي في إس**، مثلاً، المحارف

```
// $Id$
```

ستتوسع تلقائياً إلى

```
//$ Id: ClassName.java,v 1.1 2004/02/05 00:36:43 ismene Exp$
```

يسمح لك هذا بالحفاظ على معلومات إنشاء الإصدارات الحالية ضمن ملف دون أن تحتاج أي جهد تطويري آخر غير حشر تعليق Id الأصلي.

حدد الملاحظات القانونية في تعليق الكتلة تحب الكثير من الشركات أن تضمن عبارات حقوق النسخ، وملاحظات الخصوصية، والملاحظات القانونية الأخرى في برامجها. إن كان برنامجك واحداً منها، ضمن سطراً شبيهاً بالذي أسفل. افحصه مع مستشار الشركة القانوني لتحدد ما المعلومات، إن وجدت، لتضمنها في ملفاتك.

مثال جافا عن عبارة حقوق النسخ

```
// (c) Copyright 1993-2004 Steven C. McConnell. All Rights Reserved.
...
```

اعط الملف اسماً مرتبطاً بمحتواه بشكل طبيعي، ينبغي أن يكون اسم الملف قريب الارتباط باسم الصف العام المحتوى في هذا الملف. مثلاً، إن كان اسم الصف *Employee*، ينبغي أن يكون اسم الملف *Employee.cpp*. بعض اللغات، وبشكل ملحوظ **جافا**، تفرض أن يطابق اسم الملف اسم الصف.

نموذج الكتاب لتوثيق البرنامج[1]

يتفق معظم المبرمجين الخبراء على أن تقنيات التوثيق الموصوفة في القسم السابق قيّمة. الحجة العلمية المثبتة عن قيمة أي من هذه التقنيات لا تزال ضعيفة. عندما تُجمع التقنيات، على كل، الحجة مع فعاليتها تكون قوية.

في 1990، نشر باول عمان وكورتيس كوك زوجاً من الدراسات عن "نموذج الكتاب" للتوثيق (1990a،1990b). بحثوا عن أسلوب كتابة شفرة يدعم عدة أساليب مختلفة لقراءة الشفرة. كان أحد الأهداف دعم عمليات البحث من أعلى إلى أسفل، ومن أسفل إلى أعلى، والمركزة. وكان آخر أن تجزئ الشفرة إلى قطع يستطيع المبرمج تذكرها بسهولة أكثر من تعداد طويل من الشفرة المتماثلة. أراد عمان وكوك من الأسلوب أن يزود بأفكار عالية المستوى ومنخفضة المستوى معاً عن تنظيم الشفرة.

وجدوا أنه بالتفكير بالشفرة كنوع خاص من الكتب وبتنسيقها تبعاً لذلك، يستطيعون إنجاز أهدافهم. في نموذج الكتاب، تُنظم الشفرة وتوثيقها إلى عدة مكونات مشابهة لمكونات الكتاب لتساعد المبرمجين في الحصول على نظرة عالية المستوى إلى البرنامج.

---

[1] **اقرأ أيضاً** هذا النقاش تم تبنيه من "The Book Paradigm for Improved Maintenance" (عمان وكوك 1990a) و " Typographic Style Is More Than Cosmetic »

(عمان وكوك 1990b). تم تقديم تحليل مشابه مفصل في Human Factors and Typography for More Readable Programs (بيكر وماركوس 1990).

"المقدمة" هي مجموعة تعليقات استهلالية كتلك التي توجد عادة في بداية ملف. وظيفتها كمقدمة الكتاب. إنها تعطي المبرمج نظرة عامة إلى البرنامج.

يُظهر "الفهرس" الملفات والصفوف والإجرائيات عالية المستوى (الفصول). قد تظهر في لائحة، كما تظهر فصول الكتاب التقليدية، أو مرسومة في مخطط هيكلة.

"الأقسام" هي التقسيمات ضمن الإجرائيات-التصريحات عن الإجرائيات، والتصريحات عن البيانات، وعبارات تنفيذية، على سيبل المثال.

"الإشارة المرجعية" هي إشارة مرجعية تدل على شفرة، متضمنة أرقام الأسطر.

التقنيات منخفضة المستوى التي استخدمها عمان وكوك للاستفادة من ميزات التشابه بين الكتاب وتعداد الشفرة هي مشابهة للتقنيات الموصوفة في الفصل 31، "التنسيق والأسلوب،" وفي هذا الفصل.

ثمرة استخدام تقنياتهم لتنظيم الشفرة كانت عندما أعطى عمان وكوك مهمة صيانة لمجموعة من المبرمجين المحترفين الخبراء، كان الزمن الوسطي لإنجاز مهمة الصيانة في برنامج من 1000 سطر فقط حوالي ثلاثة أرباع الزمن الذي استغرقه المبرمجون ليقوموا بنفس المهمة على تعداد مصدري تقليدي (1990b). أكثر بعد، نتائج المبرمجين في صيانة شفرة موثقة باستخدام **نموذج الكتاب** كانت بشكل متوسط أعلى ب 20 بالمئة من الشفرة الموثقة بشكل تقليدي. ختم عمان وكوك بأن بالانتباه إلى المبادئ المطبعية لتصميم الكتاب، تستطيع أن تحصل على تحسين بنسبة من 10 إلى 20 بالمئة في الفهم. قدمت دراسة مع مبرمجين في جامعة تورنتو نتائج مشابهة (بيكر وماركوس 1990).

يؤكد **نموذج الكتاب** على أهمية تقديم توثيق يشرح كلا التنظيمين عالي المستوى ومنخفض المستوى لبرنامجك.

32.6 معايير آي تربل إي

من أجل التوثيق في مستوى ما وراء الشفرة المصدرية، معايير هندسة البرمجيات الخاصة ب **آي تربل إي** (معهد مهندسي الكهرباء والالكترون) هي مصادر قيّمة للمعلومات. طُوّرت معايير آي تربل إي من مجموعة مؤلفة من أصحاب المهن والأكاديميين الخبراء في مجالات محددة. يحتوي كل معيار على ملخص عن المنطقة المغطاة بالمعيار وعادة يحتوي على إطار للتوثيق المناسب للعمل في تلك المنطقة.

انضمت عدة من المنظمات الوطنية والعالمية إلى العمل على المشروع. أخذت آي تربل إي دور القيادة في تعريف معايير هندسة البرمجيات. تم تبني بعض المعايير باتحاد من قبل **آيزو** (منظمة المعايير العالمية)، أو **إيا** (اتحاد الصناعات الالكترونية)، أو **إيك** (جمعية الهندسة العالمية).

تتألف أسماء المعايير من رقم المعيار، وسنة تبني المعيار، واسم المعيار. إذاً، *IEEE/EIA Std 12207-1997, Information Technology-Software Life Cycle Processes*، يشير إلى المعيار رقم 12207.2، الذي تم تبنيه في 1997 من قبل آي تربل إي وإيا.

هاهنا بعض من المعايير الوطنية والعالمية ذات التطبيق الأكبر على المشاريع البرمجية

المعيار الأعلى مستوى هو *ISO/IEC Std 12207, Information Technology—Software Life Cycle Processes*[1]، ويكون معيار عالمي يعرّف منصة دورة الحياة لتطوير وإدارة المشاريع البرمجية. تم تبني هذا المعيار في الولايات المتحدة باسم *IEEE/EIA Std 12207, Information Technology—Software Life Cycle Processes*.

معايير تطوير البرمجيات

هاهنا معايير تطوير البرمجيات للأخذ بعين الاعتبار[2]:

IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications

تطبيقات يُنصح بها لمواصفات المتطلبات البرمجية

IEEE Std 1233-1998, Guide for Developing System Requirements Specifications

توجيه لتطوير مواصفات المتطلبات البرمجية

IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions

تطبيق يُنصح به لوصف تصميم البرمجية

IEEE Std 828-1998, Standard for Software Configuration Management Plans

معيار لمخططات إدارة الإعدادات البرمجية

IEEE Std 1063-2001, Standard for Software User Documentation

معيار لتوثيق المستخدم للبرمجيات

IEEE Std 1219-1998, Standard for Software Maintenance

معيار لصيانة البرمجيات

معايير ضمان جودة البرمجيات

[1] cc2e.com/3266

[2] cc2e.com/3273

وها هنا معايير لضمان جودة البرمجية[1]:

IEEE Std 730-2002, Standard for Software Quality Assurance Plans
معيار لمخططات ضمان جودة البرمجية

IEEE Std 1028-1997, Standard for Software Reviews
معيار لمراجعات البرمجية

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing
معيار لاختبار وحدة البرمجية

IEEE Std 829-1998, Standard for Software Test Documentation
معيار لتوثيق اختبار البرمجية

IEEE Std 1061-1998, Standard for a Software Quality Metrics Methodology
معيار لمنهجية قياسات جودة البرمجية

معايير الإدارة

هاهنا بعض معايير إدارة البرمجيات[2]:

IEEE Std 1058-1998, Standard for Software Project Management Plans
معيار لمخططات إدارة المشاريع البرمجية

IEEE Std 1074-1997, Standard for Developing Software Life Cycle Processes
معيار لتطوير عمليات دورة حياة البرمجية

IEEE Std 1045-1992, Standard for Software Productivity Metrics
معيار لقياسات إنتاجية البرمجية

IEEE Std 1062-1998, Recommended Practice for Software Acquisition
تطبيق يُنصح به للحصول على البرمجيات

---

[1] cc2e.com/3280

[2] cc2e.com/3287

IEEE Std 1540-2001, Standard for Software Life Cycle Processes - Risk Management

معيار لعمليات دورة حياة البرمجية – إدارة المخاطر

IEEE Std 1490-1998, Guide - Adoption of PMI Standard - A Guide to the Project Management Body of Knowledge

توجيه – تَبنِّي لمعيار **ابي إم آي** – توجيه لهيئات المعرفة الخاصة بإدارة المشاريع

نظرة عامة على المشاريع

هاهنا مصادر تؤمن نظرات عامة على المعايير[1]:

" مجموعة معايير هندسة البرمجيات الخاصة بـ آي تربل إي" IEEE Software[2] Engineering Standards Collection, 2003 Edition. New York, NY: Institute of Electrical and Electronics Engineers (IEEE). يحتوي هذا الإصدار الشامل على 40 من معايير **أنسي/آي تربل إي** الأحدث المتعلقة بتطوير البرمجيات حتى عام 2003. يحتوي كل معيار على إطار للوثيقة، ووصف لكل مكونات الإطار، وعرض أسباب لذلك المكون. تتضمن الوثيقة معايير لمخططات ضمان الجودة، ومخططات إدارة الإعدادات، ومستندات الاختبار، ومواصفات المتطلبات، ومخططات التأكد والصلاحية، وأوصاف التصميم، ومخططات إدارة المشاريع، وتوثيق المستخدم. الكتاب هو استقطار لخبرة مئات الناس الذين هم في قمة مجالاتهم وسيساوي أي ثمن افتراضياً. بعض من المعايير أيضاً متوفر بشكل مستقل. الكل متوفر في مجتمع الحواسيب التابع لآي تربل إي في لوس ألاميتوس، كاليفورنيا ومن *www.computer.org/cspress*.

" معايير هندسة البرمجيات: خارطة طريق المستخدم." Moore, James W. Software Engineering Standards: A User's Road Map. Los Alamitos, CA: IEEE Computer Society Press, 1997، يقدم مور نظرة عامة على معايير هندسة البرمجيات التابعة لآي تربل إي

مصادر إضافية

بالإضافة إلى معايير آي تربل إي، يتوفر العديد من المصادر الأخرى عن توثيق البرامج[3].

"قراءة الشفرة: من منظور المصدر المفتوح." Spinellis, Diomidis. *Code Reading: The Open Source Perspective*. Boston, MA: AddisonWesley, 2003، هذا الكتاب هو شرح تطبيقي لتقنيات

[1] cc2e.com/3294

[2] cc2e.com/3201

[3] cc2e.com/3208

قراءة الشفرة، متضمناً أين تجد شفرة للقراءة، ونصائح لقراءة قواعد الشفرة الضخمة، وأدوات تدعم قراءة الشفرة، والعديد من الاقتراحات الأخرى المفيدة.

SourceForge.net. لعقود، المشكلة السرمدية في تعليم تطوير البرمجيات كانت إيجاد أمثلة واقعية عن شفرة إنتاجية لتشاركها مع الطلاب[1]. يتعلم العديد من الناس بسرعة أكبر بدراسة أمثلة بالحجم الطبيعي، لكن معظم قواعد الشفرة ذات الحجم الطبيعي تُعامل كمعلومات مملوكة من الشركات التي أنشأتها. تحسّنت هذه الحالة بشكل درامي من خلال جمع الانترنت والبرمجيات مفتوحة المصدر. يحتوي موقع **سورس فورج** على شفر لآلاف البرامج في سي وسي ++ وجافا وفيجوال بيسك وابي آش ابي وبايثون والعديد من اللغات الأخرى، كلها تستطيع أن تحمله مجاناً. يستطيع المبرمجون أن يستفيدوا من الخوض عبر الشفر في الموقع ليروا أمثلة في العالم الحقيقي أكبر من قدرة **الشفرة الكاملة**، الإصدار الثاني، على العرض بأمثلته القصيرة من الشفرة. سيجد المبرمجون الأحدث، الذين لم يروا مسبقاً أمثلة واسعة عن شفرة إنتاجية، هذا الموقع قيّما بشكل خاص كمصدر لتطبيقات كتابة الشفرة الجيدة والسيئة.

Sun Microsystems[2]. "كيف تكتب تعليقات على المستندات من أجل أداة Javadoc"، 2000 متوفرة من *http://java.sun.com/j2se/javadoc/writingdoccomments/*. تصف هذه المقالة كيفية استخدام Javadoc لتوثيق برامج جافا. إنها تتضمن نصيحة مفصلة عن كيفية استخدام علامات التعليقات باستخدام أسلوب التدوين *@tag*. إنها أيضاً تتضمن العديد من التفاصيل المحددة عن كيفية كتابة التعليقات نفسها. على الأرجح إن أعراف Javadoc هي معايير التوثيق في مستوى الشفرة المطورة بالشكل الأكمل والمتاحة حالياً.

هاهنا مصادر لمعلومات عن مواضيع أخرى في توثيق البرمجيات:

McConnell, Steve. *Software Project Survival Guide*. "دليل استمرار مشروع البرمجيات" Redmond, WA: Microsoft Press, 1998. يصف هذا الكتاب التوثيق المطلوب لمشاريع إدارة الأعمال الحرجة متوسطة الحجم. يؤمن موقع الوب المرتبط به العديد من قوالب التوثيق ذات الصلة.

www.construx.com.[3] يحتوي هذا الموقع (موقع شركتي) على العديد من قوالب التوثيق، وأعراف كتابة الشفرة، والمصادر الأخرى المتعلقة بكافة جوانب تطوير البرمجيات، متضمناً توثيق البرمجية.

[1] cc2e.com/3215

أتساءل عن عدد الروائيين الكبار الذين لم يقرؤوا أعمال شخص آخر مطلقاً، وعدد الرسامين الكبار الذين لم يدرسوا حركات فرشاة غيرهم مطلقاً، وعدد الجراحين الماهرين الذين لم يتعلموا أبداً بالنظر فوق مستوى الكلية .... وبعد هذا ما نتوقع من المبرمجين أن يفعلوه.-ديف توماس

[2] cc2e.com/3222

[3] cc2e.com/3229

Post, Ed. “Real Programmers Don’t Use "المبرمجون الحقيقيون لا يستخدمون باسكال[1]" Pascal,” *Datamation*, July 1983, pp. 263–265. تناقش هذه المقالة الساخرة من أجل العودة إلى “الأيام الجميلة الماضية" لبرمجة فورتران عندما لم يكن على المبرمجين أن يهتموا بقضايا متعبة مثل قابلية القراءة.

[1] cc2e.com/3236

لائحة اختبار: تقنيات كتابة تعليقات جيدة[1]

عام

- هل يستطيع شخص ما أن يلتقط الشفرة ويبدأ بفهمها حالا؟
- هل تشرح التعليقات القصد من الشفرة أو تلخص ما تقوم به الشفرة، بدلاً من مجرد تكرار الشفرة؟
- هل استُخدمت عملية البرمجة بالشفرة الزائفة لتخفيض زمن كتابة التعليقات؟
- هل أُعيدت كتابة الشفرة المخادعة بدلاً من التعليق عليها؟
- هل التعليقات مُحدَّثة؟
- هل التعليقات واضحة وصحيحة؟
- هل يسمح أسلوب كتابة التعليقات بتعديل التعليقات بسهولة؟

العبارات والفقرات

- هل تتجنب الشفرة تعليقات نهاية السطر؟
- هل تركز التعليقات على لمَ بدلاً من كيف؟
- هل تهيّئ التعليقات الشفرة التالية للقارئ؟
- هل كل التعليقات مقبولة رسمياً؟ هل تمت إزالة أو تحسين التعليقات المكررة والغريبة والمنغمسة في اللذات؟
- هل وُثّقت المفاجئات؟
- هل اجتُنبت الاختصارات؟
- هل الفرق بين التعليقات الرئيسة والثانوية واضح؟
- هل تم التعليق على الشفرة التي تتجاوز خطأ والميزات غير المُوثَّقة؟

التصريحات عن البيانات

- هل تم التعليق على الواحدات في التصريحات عن البيانات؟
- هل تم التعليق على مجالات القيم للبيانات العددية؟
- هل تم التعليق على المعاني المرمزة؟
- هل تم التعليق على القيود على بيانات الدخل؟
- هل الرايات مُوثَّقة في مستوى البت؟
- هل تم التعليق على كل متحول شامل في مكان التصريح عنه؟
- هل تم تعريف كل متحول شامل على أنه كذلك عند كل استخدام، باستخدام عرف أو تعليق أو كليهما؟

[1] cc2e.com/3243

- هل استُبدلت الأرقام السحرية بثوابت مسماة أو متحولات بدلاً من مجرد التعليق عليها؟

بنى التحكم

- هل تم التعليق على كل عبارة تحكم؟
- هل تم التعليق على نهايات بنى التحكم الطويلة أو المعقدة، أو، عند الإمكان، بُسّطت حتى لا تحتاج إلى تعليقات؟

الإجرائيات

- هل تم التعليق على الغرض من كل إجرائية؟
- هل أُعطيت الحقائق الأخرى عن كل إجرائية في تعليقات، عند اتصالها بالموضوع، متضمنة بيانات الدخل والخرج، وافتراضات الواجهة، والقيود، وتصحيح الأخطاء، والآثار الشاملة، ومصادر الخوارزميات؟

الملفات والصفوف والبرامج

- هل يمتلك البرنامج توثيقاً قصيراً، مثل ذلك الموصوف في **نموذج الكتاب**، يعطي نظرة عامة عن كيفية تنظيم البرنامج؟
- هل وُصف الغرض من كل ملف؟
- هل اسم الكاتب، وبريده الالكتروني، ورقم هاتفه في التعداد المصدري؟

نقاط مفتاحية

- السؤال هل تُكتب التعليقات أم لا هو سؤال شرعي. بإنجازها برداءة، تكون التعليقات مضيعة للوقت وأحياناً مؤذية. بإنجازها بجودة، تستحق كتابة التعليقات الجهد المبذول من أجلها.
- ينبغي أن تحتوي الشفرة المصدرية على معظم المعلومات الحساسة عن البرنامج. طالما أن البرنامج قيد التشغيل، تكون الشفرة المصدرية مرجحة أكثر من أي مصدر آخر أن تبقى موجودة، ومن المفيد أن يكون لديك معلومات مهمة محزومة مع الشفرة.
- الشفرة الجيدة هي في توثيقها الأفضل. إن كانت الشفرة سيئة كفاية حتى تحتاج تعليقات واسعة، حاول أولاً أن تحسّن الشفرة حتى لا تحتاج تعليقات واسعة.
- ينبغي أن تقول التعليقات شيئًا عن الشفرة لا تستطيع الشفرة قوله عن نفسها-في مستوى التلخيص أو مستوى القصد.
- تحتاج بعض أساليب كتابة التعليقات للكثير من العمل الكهنوتي الممل. طور أسلوباً سهل الصيانة.

الفصل 33

الميزة الشخصية

المحتويات[1]

مواضيع ذات صلة

- مواضيع في مهنة البرمجيات: الفصل 34
- التعقيد: القسمين 5. 2 و19. 6

تلقت الميزة الشخصية درجة قليلة من الانتباه في تطوير البرمجيات. لكن منذ مقالة ادجر ديكسترا 1965 الرائدة، "اعتُبرت البرمجة كنشاط إنساني،" فقد اعتبرت شخصية المبرمج كحيّز مثمر وقانوني في التحقيق. عناوين مثل *علم نفس بناء الجسر* و*"* تجارب استكشافية في سلوك الوكيل الشرعي" قد تبدو منافية للعقل، لكن في حقل الحاسوب *علم نفس برمجة الحاسوب*، "تجارب استكشافية في سلوك المبرمج،" وعناوين مشابهة هي عناوين تقليدية.

يتعلم المهندسون في كل مجال معرفةٍ حدودَ الأدوات والمواد التي يعملون بها. إن كنت مهندس كهرباء، فإنك تعلم ناقلية العديد من المعادن ومئة طريقة لاستخدام مقياس الفولط. إن كنت مهندساً معمارياً، فإنك تعلم خصائص إسناد الأحمال للخشب والاسمنت والفولاذ.

[1] cc2e.com/3313

إن كنت مهندس برمجيات، فإن مادة البناء الأساسية هي الفكر الإنساني وأداتك الرئيسية هي *أنت*. بدلاً من تصميم هيكل إلى التفصيل الأخير ومن ثم تسليم نسخة زرقاء إلى شخص آخر من أجل البناء، فإنك تعلم أنه حالما تصمم جزء من البرمجية إلى التفصيل الأخير، تكون تمَّث. كل عمل البرمجة هو بناء قلاع في الخيال-إنه أحد النشاطات العقلية الأكثر نقاء التي يمكن أن تقوم بها.

إذن، عندما يدرس مهندسو البرمجيات الخصائص الجوهرية لأدواتهم وموادهم الخام، يجدون أنهم يدرسون الناس: الفكر، الشخصية، وصفات أخرى ملموسة أقل من الخشب والاسمنت والفولاذ.

إذا كنت تبحث عن نصائح برمجية محددة، فإن هذا الفصل يبدو مجرداً جداً حتى يكون مفيداً. وما إن تستوعب النصائح المخصصة في بقية هذا الكتاب، على كل، سيفصّل هذا الفصل لك ما تحتاج أن تفعل لتواصل التحسّن. اقرأ القسم التالي، ومن ثم قرر إن كنت تريد تجاوز هذا الفصل.

33.1 أليست الشخصية خارج الموضوع

يجعل الاهتمام الشديد بالأمور المعنوية في البرمجة من الشخصية شيئاً مهماً بشكل خاص. إنك تعلم كم هو صعب أن تعمل ثماني ساعات مركزة في يوم واحد. قد تكون مررت بتجربة الإنهاك في يوم ما بسبب التركيز الشديد جداً في اليوم الذي قبله أو الإنهاك في شهر من التركيز الشديد جداً في الشهر الذي قبله. قد تكون مرت عليك أيام كنت تعمل بها جيداً من 8:00 صباحاً إلى 2:00 بعد الظهر. وبعدها شعرت وكأنك تنطفئ. إنك لم تنطفئ، بل رغماً؛ واصلت من 2:00 بعد الظهر إلى 5:00 بعد الظهر ومن ثم قضيت بقية الأسبوع تصلح ما كتبت من 2:00 إلى 5:00.

العمل البرمجي بجوهره غير مرئي لأن أحداً لا يعرف حقيقةً ما تعمل عليه. كلنا كان لدينا مشاريع صرفنا فيها 80 بالمئة من الوقت ونحن نعمل على قطعة صغيرة وجدناها ممتعة و20 بالمئة من الوقت ونحن نبني ما تبقى من البرنامج.

لا يستطيع رب عملك أن يجبرك على أن تكون مبرمجاً جيداً، وفي كثير من المرات رب عملك ليس حتى في موقع الحكم عليك إن كنت جيداً. إن كنت تريد أن تكون عظيماً، فإنك مسؤول عن جعل نفسك عظيماً. إنها قضية شخصيتك.

بيانات مثبتة

ما إن تقرر أن تجعل من نفسك مبرمجاً متفوقاً، فإن إمكانية التحسين عملاقة. وجدت دراسة من بعد دراسة فوارق بحدود 10 إلى 1 في الوقت المطلوب لإنشاء برنامج. وجدت أيضاً فوارق بحدود 10 إلى 1 في الوقت المطلوب *لتصحيح* برنامج و10 إلى 1 في ناتج الحجم والسرعة ومعدل الأخطاء وعدد الأخطاء المكتشفة (ساكمان وإريكسون وغرانت 1068؛ كورتيس 1981؛ ميلز 1983؛ ديماركو وليستر 1985؛ كورتيس وآخرون 1986؛ كارد 1987؛ فاليت وماك غاري 1989).

لا تستطيع أن تعمل أي شيء بخصوص ذكاءك، كما تقول الحكمة التقليدية، لكن تستطيع أن تعمل شيئاً ما بخصوص شخصيتك. وهذا يكشف أن تلك الشخصية هي العامل الأكثر حسماً في بناء مبرمج متفوق.

33. 2 الذكاء والتواضع

لا يبدو الذكاء كجانب من جوانب الشخصية، وليس بجانب. صدفةً، الذكاء العظيم هو فقط متصل، بضعف لا غير، بكونك مبرمجاً جيداً.[1]

ماذا؟ لا يتوجب عليك أن تكون خارق الذكاء؟

لا، ليس عليك ذلك. لا أحد ذكي فعلياً كفاية حتى يبرمج الحواسيب. يتطلب الفهم الكامل لبرنامج متوسط تقريباً سعة غير محدودة لاستيعاب التفاصيل وسعة مساوية لتجمع شملها في نفس الوقت كلها. الطريقة التي تركز فيها ذكاءك أكثر أهمية من مقدار الذكاء الذي تملك.

كما ذكر الفصل 5 ("التصميم في البناء")، في **محاضرة جائزة تورينغ** 1972، سلّم إدجر ديكسترا مقالة بعنوان "**المبرمج المتواضع**." ناقش فيها أن معظم البرمجة هي محاولة لتعويض الحجم المحدود كثيراً لجماجمنا. الناس الذين هم الأفضل في البرمجة هم الناس الذين تحققوا من مدى صغر أدمغتهم. إنهم متواضعون. الناس الذين هم الأسوء في البرمجة هم الذين يرفضون أن يتقبلوا حقيقة كون أدمغتهم غير مكافئة للمهمة. تحجبهم "**أناهم**" من أن يكونوا مبرمجين عظماء. كلما تعلمت أكثر عن التعويض عن دماغك الصغير، أصبحت مبرمجاً أفضل. كلما كنت متواضعاً أكثر، تحسنت بسرعة أكبر.

الغرض من العديد من التطبيقات البرمجية الجيدة هو تخفيض الحمل على خلاياك الرمادية. إليك بضعة أمثلة:

- الغاية من "تحليل" نظام هي أن تجعله أبسط على الفهم. (انظر "مستويات التصميم" في القسم 5. 2 للمزيد من التفاصيل.)
- المراجعات والتفحصات والاختبارات المدارة هي طريقة للتعويض عن قابلية الخطأ المتوقعة للإنسان. أنشِـــئت تقنيات المراجعة هذه كقســـم من "البرمجة البعيدة عن الأنا" (وينبيرغ 1998). إن كنت لا ترتكب أخطاء مطلقاً، فإنك لا تحتاج إلى مراجعة برمجيتك. لكنك تعلم أن سعتك الفكرية محدودة، لذا تزيدها بسعة شخص آخر.
- الإبقاء على الإجرائيات قصيرة يخفّض الحمولة على دماغك.
- كتابة البرامج بمفردات مجال المشـكلة بدلاً من مفردات تفاصـيل التحقيق منخفض المسـتوى يخفّض من حمولتك الفكرية.

[1] نصبح حكّاماً وخبراء في الميادين التطبيقية والعلمية بالكثير جداً من أنشطة منفصلة وساعات من العمل. إن حافظ شخص بإخلاص على الانشغال في كل ساعة من يوم العمل، يستطيع أن يستيقن أنه سيستيقظ في صباح ما ليجد نفسه واحداً من الأكفاء في جيله. --ويليام جيمز

- اســتخدام الأعراف من كل الأنواع يحرر دماغك من الجوانب الدونية نســبياً للبرمجة، والتي تســبب غرامة صغيرة.

قد تظن أن الطريق الحسن سيكون أن تطور قدرات فكرية أفضل بحيث لا تحتاج إلى هذه الركائز البرمجية. قد تظن أن مبرمجاً استخدم الركائز البرمجية يسير على الطريق السيء. تجريبياً، على كل، ظهر أن المبرمجين المتواضعين الذين يعوضون عن قابلية خطأهم يكتبون شفرة تكون أسهل لهم ولغيرهم فهماً، وتمتلك أخطاء أقل. الطريق السيء الفعلي هو طريق الأخطاء والمواعيد المؤَخَّرة.

33. 3 حب الاطّلاع

متى سلمت بإن دماغك صغير جداً حتى تفهم كل البرامج ومتى تحققت من أن البرمجة الفعالة هي بحث عن طرق تُوازِن تلك الحقيقة، تكون قد بدأت بحثاً على امتداد المهنة عن طرق التعويض. في تطور المبرمج المتفوق، يجب أن يكون حب الاطلاع المتعلق بالمواضيع التقنية حتماً أولوية. المعلومات التقنية ذات الصلة تتغير باستمرار. العديد من مبرمجي **الوب** لم يضطروا للبرمجة في **مايكروسوفت ويندوز**، والعديد من مبرمجي **ويندوز** لم يضطروا مطلقاً للتعامل مع **الدوس** أو **يونيكس** أو البطاقات المثقبة. تتغير ملامح محددة للبيئة التقنية كل 5 إلى 10 سنوات. إن لم تكن محباً للاطلاع كفاية لتواكب التغيرات، قد تجد نفسك في الأسفل في بيت المبرمجين القدامى تلعب الورق مع ريكس ذو العظمة T وأخوات الديناصور آكل الأعشاب.

المبرمجون منشغلون جداً بعملهم، في معظم الأحيان لا يملكون وقتاً للفضول بخصوص كيفية القيام بأعمالهم بطريقة أفضل. إن كان هذا صحيحاً بالنسبة لك، فإنك لست وحيداً. تصف الأقسام الفرعية التالية عدة إجراءات محددة تستطيع أن تتخذها لتمرن حبك للاطلاع وتجعل التعلم أولوية.

ابنِ معرفتك حول عملية التطوير[1] كلما كنت واعياً أكثر حول بعملية التطوير، سواء أكان ذلك من القراءة أو من مراقباتك الشخصية حول تطوير البرمجيات، كلما كنت في مكان أفضل لتفهم التغيرات ولتحرك فريقك باتجاه جيد.

إن كانت أعباء عملك تتألف كلياً من وظائف قصيرة الأمد لن تتطور مهاراتك، كن غير راض. إن كنت تعمل في سوق برمجي تنافسي، نصف ما تحتاج أن تعرفه الآن لتقوم بعملك سيكون منتهي الصلاحية في ثلاث سنوات. إن لم تكن تتعلم، فإنك تتحول إلى ديناصور "مستحاثة".



إنك في حاجة ماسة لتصرف وقتاً للعمل من أجل إدارة لنفسك لا تكون فيها اهتماماتك تشغل بالك. ماعدا بعض الصعودات والهبوطات وبعض الأعمال التي تقوم بها وأنت تتنقل فوق البحار، من

[1] **إشارة مرجعية** لنقاش أتم عن أهمية سير العملية في تطوير البرمجيات انظر القسم 34. 2، "انتق طريقتك"

المتوقع أن متوسط أعداد الأعمال البرمجية المتوفرة في الولايات المتحدة سيزداد بشكل فظيع بين 2002 و 2012. من المتوقع أن تزداد أعمل تحليل الأنظمة بحوالي 60 بالمئة وأعمال مهندسي البرمجيات بحوالي 50 بالمئة. من أجل كل مجالات الأعمال الحاسوبية مجتمعة حوالي 1 مليون عمل جديد سينشئ بالإضافة إلى ال 3 مليون الموجودة حالياً (هيكر 2001، بي إل إس 2004). إن لم تستطع التعلم في وظيفتك، جِد واحدة جديدة.

جرب [1] إحدى الطرق الفعالة لتتعلم عن البرمجة هي أن تجرب بالبرمجة وعملية التطوير. إن لم تكن تعرف كيف تعمل إحدى ميزات لغتك تعمل، اكتب برنامجاً قصيراً لتطبق الميزة وترى كيف تعمل. نموذج بدئي! راقب تنفيذ البرنامج في المصحح. تكون في حال أفضل عندما تعمل على برنامج قصير لتختبر مفهوماً بالمقارنة مع حالك عندما تكتب برنامجاً كبيراً باستخدام ميزة لا تفهمها تماماً.

ماذا لو أظهر البرنامج القصير أن الميزة لا تعمل بالطريقة التي تريد؟ هذا ما كنت تريد أن تكتشف. من الأفضل أن تكتشفه في برنامج صغير من أن تكتشفه في واحد كبير. أحد مفاتيح البرمجة الفعالة هو أن تتعلم أن تصنع أخطاء بسرعة، وتتعلم منها في كل مرة. ارتكاب خطأ ليس خطيئة. بل هي الإخفاق في التعلم من خطأ.

اقرأ عن حل المشاكل [2] حل المشاكل هو النشاط اللبّي في بناء برمجيات الحاسوب. كتب هيربيرت سيمون تقريراً بسلسلة من التجارب على حل المشاكل البشرية. وجدت التجارب أن الكائنات البشرية لا تكتشف دائماً استراتيجيات ذكية لحل المشاكل بأنفسها، رغم ذلك نفس الاستراتيجيات يمكن حالاً أن تُعلَّم لنفس الكائنات هذه (سيمون 1996). المعنى هو حتى إن أردت أن تعيد اختراع العجلة، لا تستطيع أن تعتمد على أنك ستنجح. قد تعيد اختراع العجلة المربعة بدلاً.

حلل وخطط قبل أن تتصرف ستكتشف وجود توتر بين التحليل والتصرف. في نقطة ما عليك أن تنهي جمع البيانات وتتصرف. مشكلة معظم المبرمجين، على كل، ليست في تجاوز الحد في التحليل. رقاص الساعة حتى الآن والآن في جانب "التصرف" من القوس بحيث تستطيع أن تنتظر على الأقل حتى يقطع جزءاً من طريقه إلى المنتصف قبل أن تقلق حول الاستعصاء في جانب "شلل التحليل."

تعلم عن المشاريع الناجحة [3] إحدى الطرق الجيدة بشكل خاص في تعلم البرمجة هي أن تدرس عمل المبرمجين العظماء. يعتقد جون بنتلي أنه ينبغي أن تكون قادراً على الجلوس وبجانبك كأس من البراندي

---

[1] **إشارة مرجعية** تدور عدة جوانب مفتاحية للبرمجة حول فكرة التجريب. لتفاصيل، انظر "التجريب" في القسم 34. 9.

[2] **اقرأ أيضاً** كتاب جيمز أدامز *Conceptual Blockbusting (2001)* هو كتاب عظيم يعلّم حل المشاكل.

[3] cc2e.com/3320

وسيغار جيد وتقرأ برنامجاً بالطريقة التي تفعل مع رواية جيدة. قد لا يكون هذا بعيد المنال بقدر ما يبدو. معظم الناس لا يريدون أن يستخدموا وقت فراغهم ليمعنوا النظر بلائحة مصدرية بطول 500 صفحة، لكن العديد من الناس سيستمتعون بدراسة تصميم عالي المستوى والانغماس بلوائح مصدرية تفصيلية لمناطق منتقاة.

حقل هندسة البرمجيات يقوم باستخدام محدود بشكل استثنائي للأمثلة عن النجاحات والإخفاقات الماضية. إن كنت مهتماً بالمعمارية، ستدرس رسومات لويس سوليفان وفرانك لويد رايت وآي إم ابيه. قد تزور بعضاً من أبنيتهم. إن كنت مهتماً بالهندسة المعمارية، ستدرس جسر بروكلين وجسر تاكوما ناروز ومجموعة متنوعة من الأبنية الإسمنتية والفولاذية والخشبية الأخرى. ستدرس أمثلة عن النجاحات والإخفاقات في مجالك.

أشار توماس كون إلى أن أي علم ناضج فيه قسم يحتوي على مجموعة من المشاكل المحلولة المعروفة بشكل شائع كأمثلة عن أعمال جيدة في هذا المجال والتي تصلح أن تكون كأمثلة للأعمال المستقبلية (كون 1996). هندسة البرمجيات في البداية فقط للوصول إلى نضوج بهذا المستوى. في 1990، خَلُصت **لجنة علوم الحاسوب والتقنية** إلى وجود عدد قليل من حالات الدراسة الموثقة لكلا النجاحات والإخفاقات في حقل البرمجيات (سي إس تي بي 1990).

ناقشت مقالة في *التواصل في إيه سي إم* عن التعلم من حالات الدراسة لمشاكل البرمجة (من وكلانسي 1992). حقيقة أن أحدهم عليه أن يناقش حول هذا الأمر جليّة. وتلك، التي هي من أشهر الأعمدة البرمجية، "درر البرمجة،" بُنيت حول حالات دراسة المشاكل البرمجية، موحية أيضاً. وأحد أشهر الكتب في هندسة البرمجيات هو *The Mythical ManMonth*، هو حالة دراسة في إدارة البرمجة التابع لمشروع آي بي إم: أو إس\360.

مع أو بدون كتاب لحالات الدراسة في البرمجة، جد شفرة مكتوبة بأيدي مبرمجين متفوقين واقرأها. استأذن بالنظر إلى شفرة لمبرمجين تحترمهم. استأذن بالنظر إلى شفرة لمبرمجين لا تحترمهم. قارن شفراتهم، وقارن شفراتهم بشفرتك. ما هي الفوارق؟ لماذا هي مختلفة؟ أي طريقة أفضل؟ لماذا؟

بالإضافة إلى قراءة شفرة أناس آخرين، نمي رغبة بمعرفة ما يعتقد المبرمجون الخبراء بشفرتك. جد مبرمجين على مستوى العالم مستعدين ليقدموا لك انتقادهم. وأنت تستمع إلى الانتقاد، رشّح بعيداً النقاط التي يجب عليك أن ترشحها والمرتبطة بفرديتهم الشخصية وركز على النقاط التي تهم. ثم غير برمجتك بحيث تصبح أفضل.

اقرأ! رهاب التوثيق شائع بين المبرمجين. يميل التوثيق الحاسوبي ليكون مكتوباً برداءة ومنظماً برداءة، لكن مع كل مشاكله، يوجد الكثير لتكسبه من التغلب على الخوف المفرط من فوتونات شاشة الحاسوب أو المنتجات الورقية. يحتوي التوثيق على مفاتيح القلعة، وهو يستحق الوقت المصروف بقراءته. إغفال معلومة يمكن الحصول عليها حالاً هو مثال عن سهو شائع بنفس مستوى أن اختصار شائع في مجموعات الأخبار ولوحات الإعلانات هو "RTFM!" والذي يمثل "Read the !#*%@& Manual!" "اقرأ الدليل ال @#$%! "

اقرأ كتباً ومجلات أخرى[1] ربّت على ظهرك لأنك تقرأ هذا الكتاب. فإنك تعلمت مسبقاً أكثر من معظم الناس في صناعة البرمجيات لأن كتاب واحد هو أكثر مما يقرأ معظم المبرمجين في السنة (دي ماركو وليستر 1999). تقطع قراءة صغيرة شوطاً طويلاً باتجاه التقدم المهني. إن كنت تقرأ كتاباً برمجياً جيداً واحداً كل شهرين، تقريباً 35 صفحة في الأسبوع، سيكون لديك سريعاً إدراك متين في الإنتاج وستميز نفسك عن كل من حولك تقريباً.

انضم إلى محترفين آخرين جد أناس آخرين يهتمون بزيادة حدة مهاراتهم في تطوير البرمجيات. احضر مؤتمراً، انضم إلى مجموعة محلية لمستخدمي الحاسوب أو انتسب إلى مجموعة نقاش عبر الشبكة.

تعهّد بتطوير مهنتك [2] يبحث المبرمجون الجيدون دوماً عن طرق ليصبحوا أفضل. ضع في حسبانك سلم التطوير المهني التالي المستخدم في شركتي وفي عدة أُخَر:

- **المستوى 1: البداية** المبتدئ هو مبرمج قادر على ا ستخدام المقدرات الأ سا سية للغة واحدة. مثل هذا الشخص يستطيع أن يكتب صفوفاً وإجرائيات وحلقات و شرطيات ويستخدم العديد من ميزات اللغة.
- **المستوى 2: التمهيدي** المبرمج المتوســط الذي انتقل إلى ما بعد طور المبتدئ هو القادر على استخدام المقدرات الأساسية لعدة لغات وهو يرتاح جداً في لغة واحدة على الأقل.
- **المستوى 3: الكفاءة** المبرمج الكفء يمتلك خبرة في لغة أو بيئة أو كليهما. المبرمج في هذا المستوى قد يعلم كل تعقيدات J2EE أو يحفظ عن ظهر قلب "دليل سي ++ المرجعي " *Annotated C++ Reference Manual*. المبرمجين في هذا المستوى قيّمون بالنسبة لشركاتهم، والعديد من المبرمجين لا ينتقلون إلى ما بعد هذا المستوى.
- **المستوى 4: القيادة** القائد يمتلك خبرة مبرمج المستوى 3 وتَعرَّف أن البرمجة هي 15 بالمئة فقط تواصل مع الحاسوب و85 بالمئة تواصل مع الناس. فقط 30 بالمئة من وقت المبرمج الوسطي يُصرف في العمل على انفراد (ماك كو 1978). وحتى زمن أقل يُصــرف في التواصــل مع الحاســوب. القائد الروحي يكتب شــفرة من أجل جمهور الناس وليس الآلات. يكتب المبرمجون في الدرجة الصــحيحة

[1] **إشارة مرجعية** من أجل الكتب تستطيع أن تستخدم برنامج قراءة شخصي، انظر القسم 35. 4، "مخطط قراءة مطور البرمجيات."

[2] **اقرأ أيضاً** لنقاشات أخرى عن مستويات المبرمج، انظر "Construx’s Professional Development Program " (الفصل 16) في *Professional Software Development* (McConnell 2004).

للقيادة الروحية شفرة بصفاء البلّور، ويوثقونه، أيضاً. لا يريدون أن يبذّروا بخلاياهم الرمادية النفيسة القيّمة لإعادة بناء منطقِ قسمٍ من الشفرة يمكن أن يعرفوه بتعليق مكون من جملة واحدة.

كاتب الشفرة العظيم الذي لا يؤكد على موضوع سهولة القراءة قد يعلق في المستوى 3، لكن حتى هذا الأمر ليس ما يحدث عادةً. في تجربتي الشخصية، فإن السبب الرئيسي وراء كتابة الناس لشفرة غير مقروءة هو أن شفرتهم سيئة. لا يقولون لأنفسهم، "شفرتي سيئة، لذا لأجعلنّها صعبة القراءة." إنهم فقط لا يفهمون شفرتهم جيداً بما يكفي ليجعلوها مقروءة، ما يحجزهم في واحد من المستويات الأدنى.

أسوأ شفرة رأيتها في حياتي كتبتها واحدة لا تريد أن تدع أي كان يمرّ بالقرب من برامجها. أخيراً، هددها مديرها بالطرد إن لم تتعاون. كانت شفرتها خالية من التعليقات ومتسخة بمتحولات مثل *x, xx, xxx, xx1, xx2* كل منها كان شاملاً. اعتقد رئيس مديرها أنها مبرمجة عظيمة لأنها أصلحت أخطاء بسرعة. جودة شفرتها أعطتها فرصة كبيرة لتشرح قدرتها على تصحيح الأخطاء.

ليس ذنباً أن تكون مبتدئ أو متوسطاً. ليس ذنباً أن تكون مبرمجاً كفءاً وليس قائداً. الذنب هو كم تستغرق في بقائك مبتدئ أو متوسطاً بعد أن تعرف ما يجب أن تفعله لتتطور.

33.4 الأمانة الفكرية

تطوير إحساس عميق بالأمانة الفكرية هو جزء من النضوج في الاحتراف البرمجي. بشكل شائع توضح الأمانة الفكرية ذاتها بعدة طرق:

- رفض اعتبار نفسك خبيراً عندما لا تكون
- الاعتراف بأخطائك حالاً
- محاولة فهم تحذيرات المترجم بدلاً من إخفاء هذه الرسائل
- الفهم الواضح لبرنامجك -وليس ترجمته لترى إن كان يعمل
- التزويد بتقارير حالة واقعية
- التزويد بتوقعات مواعيد واقعية والثبات عليها عندما تسألك الإدارة لتعديلها

يكرر البندين الأولين في هذه اللائحة-الاعتراف بأنك لا تعرف شيئاً ما أو أنك صنعت خطأ-موضوع التواضع الفكري المُناقَش سابقاً. كيف يمكنك أن تتعلم أي شيء جديد إن كنت تعتبر أنك تعرف كل شيء مسبقاً؟ ستكون في حال أفضل إن اعتبرت أنك لا تعلم أي شيء. استمع إلى أقاويل الناس، وتعلم شيئاً جديداً منها، وقدّر إن *كانوا* يعرفون ما هم يتكلمون عنه.

استعد لتحدد كمية الثقة التي تملكها تجاه أي أمر. إن كانت عادةً 100 بالمئة، فإن هذا علامة تحذير.

رفض الاعتراف بالأخطاء هو عادة مزعجة بشكل خاص.[1] إن رفضت سالي الاعتراف بخطأ، فإنها بوضوح تصدّق أن عدم الاعتراف بالخطأ سيخدع الآخرين حتى يصدقوا أنها لم تفعله. بل العكس صحيح. كل الناس سيعلمون أنها صنعت خطأ. تُقبل الأخطاء كجزء من المدّ والجزر للنشاطات الفكرية المعقدة، وطالما هي غير مهملة، لا أحد سيمسك أخطاء عليها.

إن رفضت أن تعترف بخطأ، فإن الشخص الوحيد الذي ستخدعه هو نفسها. كل الناس ما عداها سيتعلمون أنهم يعملون مع مبرمج متكبر غير صادق بالكامل. هذا خلل يسبب اللعنة أكثر من ارتكاب خطأ بسيط. إن ارتكبت خطأ، اعترف به بسرعة وتأكيد.

الاعتبار بأنك تفهم رسائل المترجم وأنت لا تفهم هو نقطة عمياء أخرى. إن لم تفهم تحذيرات المترجم أو إن ظننت أنك تعرف ما تعني لكنك مضغوط جداً في الوقت لتختبرها، خمّن ما هو المضيعة الفعلية للوقت؟ قد تضطر إلى محاولة حل المشكلة من البداية بينما يلوّح المترجم بالحل في وجهك. قصدني الكثير من الناس ليسألوا عن مساعدة في *تصحيح* البرامج. كنت أسأل إن كانوا يقومون بترجمة نظيفة، وكان يجيبون بنعم، بعدها يبدؤون بشرح أعراض المشكلة، وأقول، "امممم، يبدو هذا وكأنه مؤشر غير مهيأ، لكن ينبغي أن يكون المترجم قد حذركم من هذا." ثم يقولون، "آ ها-لقد حذّر بالفعل من هذا. اعتقدنا أن هذا يعني شيئاً آخر." من الصعب أن تخادع الناس الآخرين عن أخطائك. وإنه لصعب أكثر أن تخادع الحاسوب، لذا لا تضييع وقتك في التجريب.

نوع ذو صلة من الإهمال الفكري يحدث عندما لا تكون فاهماً تماماً لبرنامجك و"فقط تترجمه لترى إن كان يعمل." أحد الأمثلة هو تشغيل البرنامج لترى إن كان عليك أن تستخدم < أو <=. في هذه الحالة، لا يهم فعلياً إن نجح برنامجك أم لا لأنك لست فاهماً له بشكل جيد كفاية لتعرف لم هو يعمل بنجاح. تذكر أن الاختبار يستطيع أن يُظهر فقط وجود أخطاء، وليس غيابها. إن كنت لست فاهماً للبرنامج، لا تستطيع اختباره بعمق. الإحساس الانجذاب لتترجم برنامجاً ل "ترى ما الذي يحدث" هو علامة تحذير. قد يعني هذا أنك تحتاج أن تتراجع إلى مرحلة التصميم أو أنك بدأت بكتابة الشفرة قبل التأكد من معرفة ما ستقوم بفعله. تأكد أنك تمتلك قبضة فكرية قوية على البرنامج قبل أن تتنازل عنه للمترجم.

إعطاء تقرير عن الحالة هو نطاق للازدواجية الفاضحة.[2] المبرمجون مشهورون بسوء السمعة لقولهم إن البرنامج "مكتمل بنسبة 90 بالمئة" خلال ال 50 بالمئة الأخيرة من المشروع. إن كانت مشكلتك أنه لديك إحساس

---

1 أي أحمق يستطيع أن يدافع عن خطئه—ومعظم الحمقى يفعلون. --ديل كارنيغي

2 ال 90 بالمئة الأولى من الشفرة تسبب ال 90 بالمئة الأولى من وقت التطوير. ال 10 بالمئة الباقية من الشفرة تسبب ال 90 بالمئة الأخرى من وقت التطوير. --توم كارغيل

ضعيف بتقدمك الشخصي، تستطيع حله بتعلم المزيد عن كيفية قيامك بالأعمال. لكن إن كانت مشكلتك أنك لا تقول ما في قلبك لأنك تريد أن تعطي الجواب الذي يريد مديرك سماعه، فإن هذه حكاية مختلفة. يقدّر المدير عادةً الرؤية الصادقة لحالة المشروع، حتى وإن لم تكن من الآراء التي يريد المدير سماعها. إن كانت مراقباتك مدروسة جيداً، فقدّمها بقدر ما تستطيع من تروٍّ وعلى انفراد. تريد الإدارة أن يكون لديها معلومات دقيقة لتنسق بين أنشطة التطوير، والتعاون الكامل أمر جوهري.

التخمين غير الدقيق هو قضية متعلقة بإعطاء تقارير غير دقيقة عن الحالة. [1]المشهد الاعتيادي يسير كالتالي: تسأل الإدارة بيرت كي يخمن كم من الوقت سيستغرق تطوير منتج قاعدة بيانات جديد. يتحدث بيرت إلى عدة مبرمجين، ويعالج بعض الأرقام، ويأتي بتخمين هو ثمانية مبرمجين وستة أشهر. يقول مديره، "ليس هذا ما نبحث عنه فعلياً. هل تستطيع أن تقوم به في وقت أقصر، ومع مبرمجين أقل؟" يذهب بيرت بعيداً ويفكر بالموضوع ويقرر أنه من أجل فترة قصيرة سيقطع التدريب والعطلة ويجعل كل واحد يعمل أكثر قليلاً. يعود مع تخمين هو ستة مبرمجين وأربعة أشهر. يقول مديره، "هذا عظيم. إنه مشروع منخفض الأولوية نسبياً، لذا حاول أن تحافظ عليه ضمن الفترة المخصصة بدون أي وقت إضافي لأن الميزانية لن تسمح بذلك."

الخطأ الذي ارتكبه بيرت أنه لم يدرك أن التخمينات ليست موضوعاً للتفاوض. إنه يستطيع أن يصحح التخمين ليكون أكثر دقة، لكن التفاوض مع رئيسه لن يغير الوقت الذي سيستغرقه تطوير مشروع برمجي. يقول بيل ويمير من **آي بي إم**، "وجدنا أن الناس التقنيين، بالعموم، كانوا في الحقيقة جيدين جداً في تخمين متطلبات المشروع والمواعيد. المشكلة التي كانت لديهم هي في الدفاع عن قراراتهم؛ لقد احتاجوا أن يتعلموا كيف يثبتون على آرائهم" (ويمير في ميتزغير وبودي 1996).

لن يجعل بيرت مديره في سعادة أكبر على الإطلاق بإعطائه وعداً بتسليم المشروع في أربعة أشهر وتسليمه في ستة، بالمقارنة مع إعطائه وعداً وتسليمه في ستة. سيخسر مصداقيته بتسوية النزاعات، وسيكتسب الاحترام بالصمود على تخمينه.

إن طبقت الإدارة عليك ضغطاً لتغيّر تخمينك، تأكد أن القرار النهائي للقيام بالمشروع متعلق بالإدارة: "انظروا. إليكم كم سيكلّف. لا أستطيع القول إن كان يستحق هذا السعر من منظور الشركة-هذا هو عملكم. لكن أستطيع أن أخبركم كم من الوقت سيستغرق تطوير قطعة من البرمجية-هذا هو عملي. لا أستطيع "التفاوض" على كم من الوقت سيستغرق؛ فإن هذا مثل التفاوض على كم قدم في الميل. لا نستطيع أن نتفاوض على قوانين الطبيعة. لكن، على كل، نستطيع التفاوض على جوانب أخرى من المشروع التي تؤثر بالميعاد ومن ثم نعيد تخمين الميعاد. نستطيع أن نقصي ميزات، أو نخفض الأداء، أو نطور المشروع بزيادات، أو نستخدم ناساً أقل لميعاد أبعد أو ناس أكثر لميعاد أقرب."

[1] cc2e.com/3341

إحدى المبادلات الأكثر رعباً والتي سمعت بها في حياتي كانت في محاضرة عن إدارة المشاريع البرمجية. كان المتكلم مؤلف الكتاب الأكثر بيعاً في إدارة المشاريع البرمجية. واحد من الحضور سأل، "ماذا ستفعل لو أن الإدارة طلبت منك تخميناً وأنت تعلم أنه إذا قدّمت التخمين الدقيق لهم سيقولون إنه عال جداً وسيقررون أن لا يقوموا بالمشروع؟" أجاب المتكلم عن ذلك بأنها واحدة من تلك المناطق المخادعة والتي فيها عليك أن تجعل الإدارة تؤمن بالمشروع بتقليل تقييمك له. قال إنه حالما يستثمرون في القسم الأول من المشروع، سيجتازونه إلى النهاية.

جواب خاطئ! الإدارة هي المسؤولة عن قضايا الصورة الكبيرة لتشغيل الشركة. إن كانت مقدرة برمجية محددة تستحق 250 ألف دولار من منظور الشركة وخمّنت أنها ستكلف 750 ألف دولار للتطوير، لا ينبغي أن تطور الشركة البرمجية. مسؤولية الإدارة أن تقوم بهكذا محاكمات. عندما دافع المتكلم عن الكذب بكلفة المشروع، وإخبار الإدارة أنه سيكلف أقل مما سيكلف بالحقيقة، فهو دافع عن السرقة الخفية لسلطة الإدارة. إن كنت تعتقد أن مشروعاً ما هو مشروع ممتع، افتح فتحا جديداً للشركة، أمّن تدريباً قيّماً، وما إلى ذلك. لا تستطيع الإدارة وزن هذه العوامل أيضاً. لكن خداع الإدارة لاتخاذ القرار الخاطئ يمكن حرفياً أن يكلف الشركة مئات الآلاف من الدولارات. إن كلّف ذلك عملك، تكون حصلت على ما تستحق.

33.5 التواصل والتعاون

يتعلم المبرمجون الممتازون بحق كيف يعملون ويلعبون مع الآخرين. كتابة شفرة مقروءة هي جزء من الكون لاعباً في فريق. قد يقرأ الحاسوب برنامجك كما يفعل معظم المبرمجين، لكنه ليس أفضل بكثير في قراءة الشفرة الرديئة بالمقارنة مع البشر. كتوجيه لتسهيل القراءة، أبقِ على الشخص الذي عليه أن يعدّل شفرتك في ذهنك. البرمجة هي تواصل مع مبرمج آخر أولاً وتواصل من الحاسوب ثانياً.

33.6 الإبداع والانضباط

*عندما غادرت المدرسة، ظننت أنني المبرمج الأفضل في العالم. كنت أستطيع كتابة برنامج "إكس أو" لا يُهزم، وأستخدم خمس لغات برمجة مختلفة، وأنشئ برامج ب 1000 سطر "تعمل" (حقاً!). بعدها غادرت إلى **العالم الحقيقي**. كانت مهمتي الأولى في **العالم الحقيقي** أن أقرأ وأفهم برنامج **فورتران** ب 200000 سطر وبعدها أسرّعه بمقدار الضعف. أي **مبرمج حقيقي** سيخبرك أنّ كل **كتابة الشفرة المُهيكَلة** في العالم لن تساعدك في حل مشكلة مثل هذه-إنها بحاجة لموهبة حقيقية.*

-إد بوست

من الصعب أن تشرح لخرّيج علوم حاسوب طازج لما تحتاج الأعراف والانضباط الهندسي. عندما كنت في الجامعة، أكبر برنامج كتبته كان حوالي 500 سطر من الشفرة التنفيذية. كمحترف، كتبت دزينات من الأدوات

التي كانت أصغر من 500 سطر، لكن متوسط حجم المشروع الرئيس كان بين 5000 و25000 سطر، وشاركت في مشاريع بأكثر من نصف مليون سطر من الشفرة. لا يتطلب هذا النوع من الجهد نفس المهارات بقياس أكبر، بل مجموعة جديدة كلياً من المهارات.

ينظر بعض المبرمجين المبدعين إلى الانضباط بالمعايير والأعراف على أنه خناق لإبداعهم. العكس هو الصحيح. هل تستطيع أن تتخيل موقع وب فيه كل صفحة تستخدم خطاً، وألواناً، وإزاحة خط، وأساليب صورية، وتلميحات تَنقُّل مختلفة؟ ستكون النتيجة فوضى، وليس إبداعاً. بدون المعايير والأعراف في المشاريع الكبيرة، إتمام المشروع بحد ذاته مستحيل. والإبداع حتى لا يمكن تخيُّله. لا تبدد إبداعك في أشياء لا تهمّ. أسس أعرافاً في المناطق غير الحرجة بحيث تستطيع أن تركز طاقات إبداعك على المناطق التي تدخل في الحساب.

خلال عملهم المنتهي ل 15 سنة في **مختبر هندسة برمجيات ناسا**، ماك غاري وبايرسكي أعلما أن المنهجيات والأدوات التي تؤكد على الأدب الإنساني مؤثرة بشكل خاص (1990). العديد من الناس المبدعين بمستويات عالية شديدو الأدب. "صياغة الشكل هي التحرير" كما يذهب القول. يعمل المعماريون العظماء ضمن حدود المواد الفيزيائية، والوقت، والكلفة. يفعل الفنانون العظماء ذلك، أيضاً. أي شخص تفحّص رسومات ليوناردو عليه أن يَعْجَب من انتباهه المنضبط إلى التفاصيل. عندما صمم مايكلانجلو سقف **كنيسة سيستين**، قام بتقسيمه إلى مجموعات متناظرة من الأشكال الهندسية، مثل المثلثات والدوائر والمربعات. صممه في ثلاث مناطق وفقاً للمراحل **الأفلاطونية** الثلاث. بدون هذين الهيكلة والانضباط المفروضين ذاتياً، لكانت أشكال ال 300 شخص مجرد فوضى بدلاً من أن تكون عناصر متماسكة لتحفة فنية.

تتطلب التحفة البرمجية فقط بمقدار ذلك من الانضباط. إن لم تحاول أن تحلل المتطلبات والتصميم قبل أن تبدأ بكتابة الشفرة، الكثير من تعلّمك عن المشروع سيحدث خلال كتابة الشفرة وستبدو نتيجة هذا الجهد أشبه بطبعات أصابع لطفل الثلاث سنوات من عمل فنيّ.

33.7 الخمول

يوضح الخمول نفسه بعدة طرق:[1]

- تأجيل مهمة مزعجة
- القيام بمهمة مزعجة بسرعة لإبعادها عن الطريق

[1] الخمول: الخصلة التي تجعلك تسعى مسعى عظيماً لتخفَض إنفاق الطاقة الكلي. إنه يجعلك تكتب برامج موفرة للجهد والتي سيجدها الناس الآخرين مفيدة، وتوثق ما كتبت بحيث لا ينبغي عليك أن تجيب على العديد من الأسئلة حوله. --لاري وول

- كتابة أداة لتقوم بالمهمة المزعجة بحيث لا يتوجب عليك مطلقاً أن تقوم بالمهمة من جديد

بعض هذه التوضيحات حول الخمول أفضل من بعض. الأول مفيد بصعوبة بالغة. قد تكون خضت تجربة صرف ساعات عديدة بالعبث بأعمال لا تحتاج فعلياً أن تُنجز حتى لا تضطر إلى مواجهة أعمال ثانوية نسبياً والتي لا يمكنك تجنبها. أنا أكره إدخال البيانات، والعديد من البرامج يتطلب مقداراً صغيراً من إدخال البيانات. كنت معروفاً بتأخير العمل في برنامج لأيام فقط لأؤخر المهمة الحتمية: إدخال عدة صفحات من الأرقام باليد. هذه العادة هي "الخمول الخالص." إنه يشرح نفسه مجدداً في عادة ترجمة صف لترى إن كان يعمل بحيث تستطيع أن تتجنب تمرين اختبار الصف في عقلك.

المهام الصغيرة ليست أبداً سيئة بقدر ما تبدو. إن طورت عادة القيام بها حالاً، تستطيع أن تتجنب النوع المماطل من الخمول. هذه العادة هي "الخمول المُنوَّر"-النوع الثاني من الخمول. لا تزال خمولاً، لكنك تحتال على المشكلة بصرف أصغر قدر ممكن من الوقت في شيء ما مزعج.

الخيار الثالث هو أن تكتب أداة لتقوم بالمهمة المزعجة. هذا هو "الخمول طويل الأمد." إنه بلا شك النوع الأكثر إنتاجية من الخمول (شريطة أن توفر الوقت بالحد الأقصى عن طريق كتابة أداة). في هذه السياقات، مقدار محدد من الخمول مفيد.

عندما تخطو عبر المرآة، سترى الجانب الآخر من صورة الخمول. "الاندفاع " أو "بذل الجهد" لا تمتلك اللمعان الوردي الذي كانت تملكه في صف التعلم الجسماني في المدرسة الثانوية. العجلة جهد زائد وغير ضروري. إنها تظهر أنك متحمس لكن لا تظهر أنك تقوم بإنجاز عملك. من السهل أن ترتبك بين الإشارة والتقدم، وبين المشغولية والكون منتِجاً. العمل الأهم في البرمجة الفعالة هو التفكير، ويميل البشر لألا يبدون مشغولين عندما يكونون يفكرون. إن أنا عملت مع مبرمج يبدو مشغولاً كل الوقت، لكنت سأفترض أنه ليس مبرمجاً جيداً لأنه لم يكن يستخدم أداته الأكثر قيمة، دماغه.

33.8 ميزات لا تهم بقدر ما تظن

الاندفاع ليس المميزة الوحيدة التي قد تُعجب بها في جوانب أخرى من حياتك لكنها لا تعمل بشكل جيد جداً في تطوير البرمجيات.

الإصرار

حسب الحالة، قد يكون الإصرار إما شيئاً ثميناً أو عائقاً. مثل معظم المفاهيم المُحمَّلة بالقيم، فإنها تُعرَّف بكلمات مختلفة حسب الاعتقاد بأنها خصلة جيدة أو واحدة سيئة. إن كنت تريد أن تُعرِّف الإصرار كخصلة سيئة، تقول إنه "عناد" أو "صعوبة مراس." إن كنت تريد أن يكون خصلة جيدة، تدعوها "صلابة" أو "مثابرة."

في معظم الأوقات، الإصرار في تطوير البرمجيات هو صعوبة مراس-يمتلك قيمة صغيرة. الإصرار عندما تعلق في قطعة من الشفرة الجديدة هو على أصعب ما يكون فضيلة. جرّب إعادة تصميم الصف، جرّب نهج كتابة شفرة مختلفاً، جرّب أن تعود إليها لاحقاً. عندما لا يعمل أحد النُهج، يكون الوقت مناسباً لتجرّب بديلاً (بيرسيغ 1974).

في *التصحيح*[1]، قد يكون إلقاء القبض على خطأ كان يزعجك من أربع ساعات مرضياً بشكل هائل، لكن غالباً من الأفضل أن تستلم أمام الخطأ بعد مقدار محدد من الوقت لا تحرز فيه أي تقدم-لنقل 15 دقيقة. دع وعيك الباطن يمضغ المشكلة لفترة. جرّب أن تفكر بنهج بديل يطوّق المشكلة كلياً. أعد كتابة القسم الحاوي على المشاكل من الشفرة من الصفر. عُد إليه لاحقاً عندما يكون دماغك منتعشاً. مقاتلة مشاكل الحاسوب ليست فضيلة. تجنبها أفضل.

من الصعب أن تعرف متى تستلم، لكن من المهم جداً أن تسأل. عندما تلاحظ أنك محبط، يكون الوقت مناسباً لتسأل. السؤال لا يعني بالضرورة أنه وقت الاستسلام، لكنه قد يعني أنه وقت وضع بعض المحددات للنشاط: "إن لم أحل المشكلة باستخدام هذا النهج خلال ال 30 دقيقة القادمة، سأقوم بعصف ذهني ل 10 دقائق حول النهج المختلفة وأجرّب أفضل واحد في الساعة التالية."

الخبرة

قيمة تناقل الخبرة أصغر من قيمة التعلم من كتاب في تطوير البرمجيات بالمقارنة مع العديد من المجالات الأخرى لعدة أسباب. في العديد من المجالات الأخرى، تتغير المعرفة الأساسية ببطء كاف حتى يكون أحد ما تخرج في الكلية بعدك ب 10 سنين تعلّم، ربما، نفس المادة الأساسية التي تعلمت. في تطوير البرمجيات، حتى المعرفة الأساسية تتغير بسرعة. شخص تخرج في الكلية بعدك ب 10 سنين يكون تعلم، ربما، بمقدارك مرتين عن تقنيات البرمجة الفعالة. يميل المبرمجون القدامى لأن يُنظر إليهم على أنهم متهمين ليس فقط لأنهم قد يكونون بعيدين عن التماس بتقنيات محددة لكن لأنهم قد لا يتعرضون أبداً لمفاهيم برمجية أساسية أصبحت معروفة جداً بعد مغادرتهم المدرسة.

في حقول أخرى، ما تتعلم عن عملك اليوم سيساعدك على الأرجح في عملك غداً. في البرمجيات، إن لم تقدر أن تقلع عن عادات التفكير التي طورتها أثناء استخدامك لغتك البرمجية الأولى أو تقنيات معايرة الشفرة التي عملت بنجاح على آلتك القديمة، ستكون خبرتك أسوأ مما لو لم يكن لديك خبرة مطلقا. الكثير من ناس البرمجة يصرفون وقتهم بالتحضير للقتال في الحرب الأخيرة بدلاً من الحرب القادمة. إن لم تقدر أن تتغير مع الأوقات، فإن الخبرة لعائق أكثر من كونها مساعد.

[1] **إشارة مرجعية** لنقاش مفصل عن الإصرار في *التصحيح*، انظر "نصائح لإيجاد العيوب" في القسم 23. 2

علاوة على التغيرات السريعة في تطوير البرمجيات، غالباً ما يرسم البشر الخلاصات الخاطئة من تجاربهم. من الصعب أن ترى حياتك الخاصة بموضوعية. قد تغفل عن عناصر مفتاحية في خبرتك الشخصية والتي تسبب أن ترسم نتائج مختلفة عما ستفعل لو لاحظتها. قراءة دراسات مبرمجين آخرين مساعدِة لأن الدراسات تجلّي خبرات الناس الآخرين-مُرشَّحة كفاية حتى تتمكن من فحصها بموضوعية.

يضع البشر تأكيداً غير معقول على *كمية* الخبرة التي يمتلكها المبرمجون. "نريد مبرمجاً بخبرة خمس سنين من البرمجة بلغة **سي**" عبارة سخيفة. إن لم يتعلم مبرمج **سي** بعد سنة أو اثنتين، لن تصنع السنين الثلاثة التالية فرقاً ذا أهمية. لهذا النوع من "الخبرة" صلة ضعيفة بالأداء.

حقيقة أن المعلومات تتغير بسرعة في البرمجة تسير باتجاه قوى غريبة الأطوار في منطقة "الخبرة." في العديد من المجالات، محترف كان لديه تاريخ من الإنجازات يستطيع أن يخفض من أدائه، ويسترخي ويستمتع بالاحترام المكتسب من سلسلة من النجاحات. في تطوير البرمجيات، أي شخص يخفض من أدائه سيصبح بسرعة خارج الجو. لتبقى ذا قيمة، عليك أن تبقى حاضراً. للمبرمجين الجدد التواقين، إن هذا امتياز. أحياناً يشعر المبرمجون الأقدم أنهم قد اكتسبوا راياتهم من قبل ويمتعضون من الالتزام بإثبات أنفسهم سنة بعد سنة.

الخلاصة في الخبرة هي هذا: إن عملت ل 10 سنين، فهل ستحصل على 10 سنين من الخبرة أو هل ستحصل على 1 سنة من الخبرة 10 مرات؟ عليك أن تتفكر بنشاطاتك لتحصل على خبرة صحيحة. إن جعلت من التعلم عهداً مستمراً، ستحصل على الخبرة. إن لم تفعل، لن تحصل، ولا يهم عدد السنين التي كنت تعمل بها.

البرمجة الشاذة

إن لم تصرف على الأقل شهراً وأنت تعمل على نفس البرنامج-تعمل 16 ساعة من اليوم، وتحلم به خلال الساعات ال 8 الباقية من النوم القلق، وتعمل عدة ليال متتالية وأنت تحاول باستقامة أن تقصي تلك "الثغرة الأخيرة الباقية" من البرنامج-إذن لم تكتب فعلياً برنامج حاسوب معقداً. وقد لا تكون امتلكت الإحساس بوجود شيء بهيج في البرمجة.

إداورد يوردون

هذا الإجلال العظيم للساديّة البرمجية هي عقد بيع نقي وتقريباً وصفة لا شك فيها للفشل. فترات كل الليل البرمجية هذه تجعلك تشعر أنك المبرمج الأعظم في العالم، لكن بعدها عليك أن تصرف عدة أسابيع وأنت تصحح العيوب التي ركّبتها خلال توهجك بالمجد. بكل المعاني، كن متحمساً للبرمجة. لكن الحماس ليس بديلاً للكفاءة. تذكر أي منهما أهم.

33. 9 العادات

الفضائل الأخلاقية، حينئذ، تنشأ فينا ليس ب ولا بعدم مخالفة الطبيعة ..تطورها الكامل فينا هو نتيجة العادة.... نتعلم أي شيء يجب علينا أن نتعلم كيفية القيام به عن طريق القيام به فعلاً.... سيصبح الرجال بنائين جيدين كنتيجة للبناء بشكل جيد وبنائين سيئين كنتيجة للبناء بشكل سيء.... لذا فإن أي نوع من العادات تُكوِّن من السن الأصغر ليست قضية قليلة الأهمية-إنها تصنع اختلافاً شاسعاً، بل كل الاختلاف في العالم.

--أرسطو

تَهمّ العادات الجيدة، لأن معظم ما تقوم به كمبرمج تقوم به بدون أن تفكر بوعي به. مثالاً، قد تكون فكرت بالكيفية التي تريدها لتنسيق بادئات الحلقات، في إحدى المرات، لكنك الآن لا تفكر بها. مجدداً في كل مرة تكتب حلقة جديدة. فإنك تقوم بذلك بالطريقة التي تقوم بها بحكم العادة. هذا صحيح فعلياً لكل جوانب تنسيق البرنامج. متى كانت آخر مرة تساءلت بشكل جاد عن أسلوبك في التنسيق؟ توجد فرص جيدة ليكون ذلك منذ أربع سنين ونصف، إن كنت تبرمج في السنوات الخمسة الماضية. في بقية الوقت اعتمدت على العادة.

لديك عادات في الكثير من المناطق. [1]مثالاً، يميل المبرمجون لفحص فهارس الحلقة بعناية وليس لفحص عبارات الإسناد، ما يجعل الأخطاء في عبارات الإسناد أصعب للإيجاد من الأخطاء في فهارس الحلقات (غولد 1975). أنت تستجيب للانتقادات بطريقة ودية أو بطريقة غير ودية. أنت دائماً تبحث عن طرق لتجعل الشفرة مقروءة وسريعة، أو لا تفعل. إن كان عليك الاختيار بين جعل الشفرة سريعة وجعلها مقروءة، واتخذت نفس الخيار في كل مرة، فإنك لا تختار حقيقةً-أنت تستجيب للعادة.

ادرس مقولة أرسطو واستبدل "الفضائل البرمجية" ب "الفضائل الأخلاقية." لقد أشار إلى أنك لست ميالاً إلى إما السلوك الجيد أو السلوك السيء لكنك مُركَّب بطريقة تُمكِّنك من أن تصبح إما مبرمجاً جيداً أو مبرمجاً سيئاً. ما تفعله يصبح عادة، ومع مرور الوقت العادات الجيدة أو السيئة تحدد كونك مبرمجاً جيداً أو سيئاً.

يقول بيل غيتس أن أي مبرمج سيصبح في أي وقت جيداً هو جيد في السنين القليلة الأولى. بعد ذلك، سواء أكان المبرمج جيداً أو لا فقد أصبح صبّة إسمنتية (لاميرز 1986). بعد أن تبرمج لفترة طويلة، من الصعب أن تبدأ القول فجأة، "كيف أجعل هذه الحلقة أسرع؟" أو "كيف أجعل هذه الشفرة أكثر قابلية للقراءة؟" هذه عادات يطورها المبرمجون الجيدون مبكراً.

عندما تتعلم شيئاً ما لأول مرة، تعلّمه بالطريقة الصحيحة. عندما تقوم به لأول مرة، فإنك لا تزال تفكر بنشاط به ولا تزال تملك خياراً سهلاً بين القيام به بطريقة جيدة والقيام به بطريقة سيئة. بعد أن تتمه عدة مرات، فإنك تصرف انتباهاً أقل إلى ما تقوم بفعله وتسود "قوة العادة". تأكد من أن العادات التي تسود هي العادات التي تريد أن تمتلكها.

[1] **إشارة مرجعية** لتفاصيل عن الأخطاء في عبارات الإسناد، انظر "أخطاء حسب التصنيف" في القسم 22. 4

إن لم تكن قد امتلكت العادات الأكثر فعالية مسبقاً؟ كيف تُغيِّر عادة سيئة؟ إن كان لدي جواب حتمي لهذا، سأتمكن من أن أبيع أشرطة "مساعدة ذاتية" لعروض التلفاز في ساعات الليل الأخيرة. لكن هاهنا على الأقل جزء من الجواب. لا تستطيع أن تستبدل لا عادة بعادة سيئة مطلقاً. هذا سبب أن الناس الذين يتوقفون فجأة عن التدخين أو الأيمان أو الأكل الزائد يقضون وقتاً صعباً مالم يستبدلون شيئاً آخر، مثل العلكة. أن تستبدل عادة جديدة بعادة قديمة أسهل من أن تتخلص من عادة كلياً. في البرمجة، حاول أن تطور عادات جيدة تعمل بنجاح. طور عادتي كتابة الصف بالشفرة الزائفة قبل كتابته بالشفرة، وقراءة الشفرة بحذر قبل ترجمتها، مثالاً. ليس عليك أن تقلق بشأن خسارة عادات سيئة؛ ستسقط بشكل طبيعي على جانب الطريق بينما تأخذ العادات الجديدة مكانها.

مصادر إضافية

فيما يلي مصادر إضافية للجوانب الإنسانية في تطوير البرمجيات[1]

Dijkstra, Edsger. “The Humble Programmer.” Turing Award Lecture. *Communications of the ACM 15*, no. 10 (October 1972): 859–66.[2]

"المبرمج المتواضع" هذه مقالة قديمة ساعدت في بدء التحقيق في كيفية اعتماد برمجة الحاسوب على قدرات المبرمج العقلية. شدد دكسترا باستمرار على الرسالة: المهمة الرئيسة للبرمجة هي السيطرة على التعقيد الشاسع في علم الحاسوب. وبيّن أن البرمجة هي النشاط الوحيد الذي على الانسان فيه أن يسيطر على كمية من الأس التاسع من الفرق بين المستوى الأدنى من التفصيل والأعلى. قراءة هذه المقالة ممتعة فقط لقيمتها التاريخية، لكن العديد من مواضيعها تبدو جديدة بعد عقود من الزمن. وهي تبلِّغ معنى جيداً عن ماهية المبرمج في الأيام الأولى من علم الحاسوب.

Weinberg, Gerald M. The Psychology of Computer Programming: Silver Anniversary Edition. New York, NY: Dorset House, 1998.

" علم نفس برمجة الحاسوب: الطبعة السنوية الفضية" هذا الكتاب القديم يحتوي على توضيح تفصيلي لفكرة البرمجة *البعيدة عن الأنا* وللعديد من الجوانب الأخرى من الناحية الإنسانية لبرمجة الحاسوب. إنه يحتوي على العديد من الطرف الممتعة وواحد من الكتب الأكثر قراءة حتى الآن المكتوبة في مجال تطوير البرمجيات.

Pirsig, Robert M. Zen and the Art of Motorcycle Maintenance: An Inquiry into Values. William Morrow, 1974.

[1] cc2e.com/3327
[2] cc2e.com/3334

" زين وفن صيانة الدراجات النارية: تحقيق في القيم" قدّم بيرسيغ نقاشاً موسّعاً عن "الجودة،" ظاهرياً بارتباطها بصيانة الدراجة النارية. كان بيرسيغ يعمل ككاتب برمجيات تقنية عندما كتب ***زام*** (*ZAMM*)، وتعليقاته المبصرة تُطبَّق على علم نفس المشاريع البرمجية بمقدار ما تُطبق على صيانة الدراجات الآلية.

`Curtis, Bill, ed. Tutorial: Human Factors in Software Development. Los Angeles, CA: IEEE Computer Society Press, 1985.`

" العوامل البشرية في تطوير البرمجيات" هذه تجميعة ممتازة من المقالات التي تعالج الجوانب الإنسانية في إنشاء برامج حاسوبية. المقالات ال 45 مقسمة إلى أقسام على وفق: النماذج الفكرية للمعرفة البرمجية، وتعلم البرمجة، وحل المشاكل والتصميم، وآثار تمثيل التصميم، ومميزات اللغة، وتشخيص الأخطاء، والمنهجيات. إن كانت البرمجة واحدة من أصعب التحديات الفكرية التي واجهها النوع البشري على مر العصور، فإن تعلم المزيد عن السعات العقلية البشرية أمر حاسم لنجاح المسعى. هذه المقالات عن العوامل النفسية تساعدك أيضاً بقلب عقلك نحو الداخل وتعلم كيف تتمكن باستقلالية أن تبرمج بشكل أكثر فعالية.

`McConnell, Steve. Professional Software Development. Boston, MA: Addison-Wesley, 2004.`

"تطوير البرمجيات المهني" يقدم الفصل 7، "الأيتام مفضَّلون" ("Orphans Preferred") تفاصيل أكثر عن خصوصيات المبرمج ودور الشخصية.

نقاط مفتاحية

- تؤثر شخصيتك مباشرة بقدرتك على كتابة برامج حاسوبية.
- المميزات التي تهم أكثر هي التواضــع، وحب الاطلاع، والأمانة الفكرية، الإبداع والانضــباط، والخمول المُنوَّر.
- مميزات المبرمج المتفوق غير متصلة بالموهبة ومتصلة كل الصلة بالالتزام بتطوير الشخصية.
- بشكل مثير للدهشة، (الذكاء والخبرة والإصرار والجرأة) الخام تؤذي بقدر ما تساعد.
- لا يبحث العديد من المبرمجين بنشـــاط عن المعلومات والتقنيات و بدلاً يعتمدون على التعرض التصادفي والمعتمد على العمل للمعلومات الجديدة. إن كرست نسبة صغيرة من وقتك للقراءة والتعلم عن البرمجة، بعد بضعة شهور أو سنين ستميز نفسك بشكل هائل عن مجرى البرمجة الرئيس.
- الشخصية الجيدة هي بشكل رئيس موضوع امتلاك العادات الصحيحة. لتكون مبرمجاً عظيماً، طور العادات الصحيحة وستأتي البقية بشكل طبيعي.

الفصل 34

موضوعات في مهنة البرمجيات

المحتويات[1]

مواضيع ذات صلة

- كامل الكتاب

هذا الكتاب بمعظمه عن تفاصيل بناء البرمجيات: صفوف عالية الجودة، وأسماء المتحولات، والحلقات، وتخطيط الشفرة المصدرية، وتكامل النظام، وهلم جراً. ألغى هذا الكتاب التأكيد على المواضيع المجردة ليؤكد على مواضيع أكثر تحديداً.

حالما تجلّي الأقسام السابقة من الكتاب حقيقة المواضيع المحددة، كل ما عليك فعله لتدرك المفاهيم المجردة إدراكاً كاملاً هو أن تنتقي مواضيع من الفصول المختلفة وترى كيف ترتبط ببعضها. يجعل هذا الفصل الموضوعات المجردة واضحة: التعقيد، والتجريد، والطريقة، وقابلية القراءة، والتكرار، وهلم جراً. هذه الموضوعات مسؤولة عن قسم كبير من الفرق بين التعدي ومهنة البرمجيات

[1] cc2e.com/3444

34. 1 انتصر على التعقيد

يقع الدافع لتخفيض التعقيد في قلب تطوير البرمجيات.[1] إلى درجة مثل الدرجة التي بها وصف الفصل 5، "التصميم في البناء" إدارة التعقيد بالإلزام التقني الرئيس في البرمجية. مع أن محاولة أن تكون بطلاً وأن تتعامل مع مشاكل علوم الحاسوب في كل المستويات أمر مغرٍ، إلا أنه لا أحد لديه مخ قادر فعلياً على عبور طبقات الحجم التسعة للتفاصيل. طوّرت علوم الحاسوب وهندسة البرمجيات عدة أدوات فكرية للتعامل مع هكذا تعقيد، ونقاشات المواضيع الأخرى في هذا الكتاب مسحت الغبار عن الكثير منها:

- تقسيم نظام إلى أنظمة فرعية في مستوى الهيكلة بحيث يستطيع دماغك التركيز على مقدار أصغر من النظام في المرة الواحدة.
- تعريف واجهات الصفوف بحذر بحيث تستطيع تجاهل الأعمال الداخلية في الصف.
- الحفاظ على التجريد المُمثَّل بواجهة الصف بحيث لا يتوجب على دماغك أن يتذكر تفاصيل اعتباطية.
- تجنُّب البيانات الشاملة، لأن البيانات الشاملة تزيد بشكل واسع من النسبة المئوية للشفرة التي تحتاج أن تستضيفها جميعاً في دماغك في أي مرة.
- تجنب الهرميات الوراثية العميقة لأنها كثيرة المطالب فكرياً.
- تجنب التعشيش العميق للحلقات والشرطيات لأنه من الممكن أن تحل بنى تحكم أبسط محلها والتي تسبب ارتفاع حرارة عدد أقل من الخلايا الرمادية.
- تجنب *gotos* لأنها تقدم عدم خطية وُجِدَ أنها صعبة التتبع لمعظم الناس.
- تعريف نهجك للتعامل مع الأخطاء بحذر بدلاً من استخدام توالد اعتباطي لتقنيات معالجة أخطاء مختلفة.
- الكون متّبعاً للقواعد بخصوص استخدام آليات الاستثناءات المبنية أساساً في اللغة، والتي يمكن أن تكون بنيان تحكم غير خطي وتقريباً بصعوبة فهم *gotos* إن لم تستخدم بانضباط.
- عدم السماح للصفوف بأن تنمو لتصبح صفوف عملاقة بقدر برنامج كامل بحد ذاتها.
- الإبقاء على الإجرائيات قصيرة.
- استخدام أسماء متحولات واضحة وتشرح نفسها بحيث لا يتوجب على دماغك أن يضيّع دورات وهو يتذكر تفاصيل مثل "*i* يمثل فهرس الحساب، و *j* يمثل فهرس الزبون، أم كان الترتيب بالعكس؟"
- تقليل عدد الوسطاء الممررة إلى إجرائية، أو، أهم أكثر، تمرير الوسطاء اللازمة فقط للحفاظ على تجريد واجهة الإجرائية.

[1] **إشارة مرجعية** لتفاصيل حول أهمية الموقف في الانتصار على التعقيد انظر القسم 33. 2، "الذكاء والتواضع."

- اسـتخدام أعراف لتجنب دماغك من تحدي تذكُّر فوارق اعتباطية عَرَضـية بين الأقسـام المختلفة من الشفرة.
- بالعموم، مهاجمة ما وصفه الفصل 5 ب "الصعوبات العرضية" في أي وقت ممكن.

عندما تضع اختباراً معقداً في تابع منطقي وتجرِّد غرض الاختبار، تكون جعلت الشفرة أقل تعقيداً. عندما تستبدل بحث في جدول بسلسلة معقدة من المنطق، تكون فعلت نفس الشيء. عندما تنشئ واجهات صفوف معرفة بجودة ومتماسكة، تكون استغنيت عن الحاجة لتقلق بخصوص تفاصيل تحقيق الصف وبسّطت عملك كله.

الهدف من وجود أعراف كتابة الشفرة هو أيضاً تخفيض التعقيد بالدرجة الأولى. عندما تستطيع أن تمعير قرارات تخص التنسيق والحلقات وأسماء المتحولات وتراميز بناء الوحدات وما إلى ذلك، فإنك تحرر مصادر فكرية تحتاجها لتركز على جوانب أكثر تحدياً من المشكلة البرمجية. أحد الأسباب وراء الجدلية الشديدة لأعراف كتابة الشفرة هو أن للاختيارات بين الخيارات قاعدة جمالية محدودة نوعاً ما لكنها جوهرياً كيفية. تكون النقاشات الأسخن بين الناس على الفوارق الصغرى فيما بينهم. تكون الأعراف في الحالة الأكثر فائدة عندما تجنِّبك من مشاكل اتخاذ قرارات كيفية والدفاع عنها. إنها أقل قيمة عندما تفرض قيوداً في مناطق ذات معنى أكثر.

التجريد بأشكاله المتنوعة هو أداة قوية بشكل خاص لإدارة التعقيد. تقدمت البرمجة بشكل كبير عبر زيادة الحالة التجريدية لمكونات البرنامج. بيّن فريد بروكس أن أكبر كسب مفرد صُنع في تاريخ علم الحاسوب كان في القفزة من لغة الآلة إلى لغات المستوى العالي-لقد حررت المبرمجين من القلق بخصوص الشذوذات التفصيلية لقطع مستقلة من العتاد الصلب وسمحت لهم بالتركيز على البرمجة (بروكس 1995). فكرة الإجرائيات كانت خطوة كبيرة أخرى، أُتبِعت بالصفوف والحزم.

تسمية المتحولات بشكل فعال، على "ماذا" المتعلقة بالمشكلة بدلاً من "كيف" المتعلقة بالحل في مستوى التحقيق، تزيد من مستوى التجريد. إن قلت، "حسناً، أنا أفرغ المكدس وهذا يعني أنني أحصل على الموظف الأحدث،" يمكن للتجريد أن يوفر عليك الخطوة الفكرية "أنا أفرغ المكدس." تقول ببساطة، "أنا أحصل على الموظف الأحدث." هذا كسب صغير، لكن إن كنت تحاول ان تخفض مجال التعقيد الذي هو من 1 إلى 10^9، كل خطوة تُحتسَب. استخدام الثوابت المسماة بدلاً من "الحرفيات" يزيد أيضاً من مستوى التجريد. تؤمن البرمجة غرضية التوجه مستوىً من التجريد يتطبق على الخوارزميات والبيانات بنفس الوقت، وهو نوع من التجريد لا يؤمنه التحليل الوظيفي بمفرده.

بإيجاز، الهدف الرئيس من تصميم وبناء البرمجية هو الانتصار على التعقيد. الدافع وراء العديد من التطبيقات البرمجية هو تخفيض تعقيد البرنامج، وبدعم الدلائل، تخفيض التعقيد هو أهم مفتاح لتكون مبرمجاً فعالاً.

34. 2 انتق طريقتك

مجرى رئيس ثان في هذا الكتاب هو فكرة أن الطريقة التي تستخدمها لتطور برمجية تَهُم بمقدار مدهش. في المشاريع الصغيرة، ملَكات المبرمج المفرد هي المؤثر الأكبر على جودة البرمجية. جزءٌ مما يجعل المبرمج المفرد ناجحاً هو اختياره للطرق.

في المشاريع التي تضم أكثر من مبرمج، المميزات التنظيمية تصنع فارقاً أكبر مما تفعل مهارات الأفراد المتضمنين. حتى إن ملكت فريقاً عظيماً، فإن قدرته الجماعية ليست ببساطة مجموع قدرات أعضاء الفريق الفردية. تحدد الطريقة التي يعمل بها الناس مع بعضهم إن كانت قدراتهم ستضاف إلى أو تطرح من بعضها البعض. تحدد الطريقة التي يستخدمها الفريق إن كان عمل فرد يدعم عمل بقية الفريق أو يضعفه.

أحد الأمثلة على الطريقة التي بها تهم الطريقة هو تبعات ألا تصنع متطلبات مستقرة قبل البدء بالتصميم وكتابة الشفرة.[1] إن لم تكن تعلم ماذا تبني، فإنك لن تنشئ تصميماً متفوقاً له. إن تغيرت المتطلبات وتبعاً التصميم بينما تكون البرمجية تحت التطوير، يجب حتماً أن تتغير الشفرة أيضاً، والذي قد يسبب خطر هبوط جودة النظام.

"بالطبع،" تقول، "لكن في العالم الحقيقي، لن تمتلك مطلقاً متطلبات مستقرة، لذا فإن هذا يبعدنا عن السياق." مجدداً، تحدد الطريقة التي تستخدمها كم هي متطلباتك مستقرة وكم تلزم أن تكون مستقرة. إن أردت أن تبني مرونة أكثر في المتطلبات، تستطيع أن تحضّر لنهج التطوير التزايدي والذي به تخطط أن تسلّم البرمجية في عدة زيادات بدلاً من الكل دفعة واحدة. هذا انتباه إلى الطريقة، والطريقة التي تستخدمها هي التي تحدد بشكل نهائي إن كان مشروعك سينجح أو يفشل. يبيّن الجدول 3-1 في القسم 3. 1 فكرة أن أخطاء المتطلبات مكلفة أكثر بكثير من أخطاء البناء، لذا التركيز على ذلك القسم من العملية يؤثر أيضاً على الكلفة والميعاد.

نفس مبدأ الاعتناء الواعي بالطريقة يُطبَّق على التصميم. عليك أن تضع أساسات صلبة قبل أن تبدأ البناء عليها.[2] إن عَجَلت إلى التشفير قبل أن تكتمل الأساسات، سيكون القيام بتغيرات أساسية في هيكلة النظام أصعب. سيقوم الناس باستثمار عاطفي للتصميم لأنهم قد كتبوا مسبقاً شفرة من أجله. من الصعب أن ترمي بعيداً أساساً سيئاً مذ بدأت بناء بيت عليه.

السبب الرئيس في أن الطريقة تَهُم هو أنه في البرمجية، يجب حتماً أن تُبنى الجودة من الخطوة الأولى وباتجاه الأمام. هذا يتحدى الحكمة الشعبية: تستطيع أن تكتب شفرة مثل الجحيم ثم تختبر كل الأخطاء وتطردها من البرمجية. هذه الفكرة خاطئة تماماً. يخبرك الاختبار فقط بالطرق المحددة التي تكون برمجيتك فيها مختلّة. لن

[1] **إشارة مرجعية** لتفاصيل حول جعل المتطلبات مستقرة، انظر القسم 3. 4، "الشرط المسبق للمتطلبات." لتفاصيل حول التنوعات في نُهج التطوير انظر القسم 3. 2، "حدد نوع البرمجية التي تعمل عليها."

[2] رسالتي إلى المبرمج الجادّ هي: اصرف جزءاً من يوم عملك بتفحص وتنقية منهجياتك الخاصة. حتى ولو كان المبرمجون يكافحون دائماً ليصلوا إلى موعد نهائي ما في الماضي أو المستقبل، فإن الإزالة المنهجية هي استثمار حكيم طويل الأمد. --روبرت دبليو. فلويد

يجعل الاختبار برنامجك أكثر سهولة في الاستخدام، ولا أسرع ولا أصغر ولا أكثر قابلية للقراءة ولا قابلاً للتجديد أكثر.

المعايرة غير الناضجة نوع آخر من خطأ الطريقة. في طريقة فعالة، تقوم بتعديلات خشنة في البداية وتعديلات ناعمة في النهاية. إن كنت نحاتاً، ستجلف الشكل العام قبل أن تبدأ بتهذيب الملامح. تضيّع المعايرة غير الناضجة الوقت لأنك تصرف وقتاً في تهذيب أقسام من الشفرة لا تحتاج أن تُهذَّب. قد تهذب أقساماً صغيرة كفاية وسريعة كفاية كما هي، قد تهذب شفرة وترميها لاحقاً، وقد تفشل في رمي شفرة سيئة بعيداً لأنك صرفت وقتاً في تهذيبها. كن دائما في حالة تفكير، "أأقوم بهذا في الترتيب الصحيح؟ أيصنع تغير الترتيب فرقاً؟" اتبع بوعي طريقة جيدة.

سير العمليات منخفضة المستوى يهمّ أيضاً.[1] إن اتبعت طريقة كتابة شفرة زائفة ومن ثم ملئ الشفرة حول الشفرة الزائفة، فإنك تحصد منافع التصميم من الأعلى فنزولاً. وإنك أيضاً ضمنت أن يكون لديك تعليقات في الشفرة بدون أن يكون عليك أن تكتبها لاحقاً.

مراقبة سير عمليات ضخمة وسير عمليات صغيرة تعني التوقف للانتباه على كيفية إنشائك للبرمجية. إنه وقت مصروف بمحله. قولك إن "الشفرة هي ما يهمّ؛ عليك أن تركز على كم هي الشفرة جيدة، وليس على سير بعض العمليات المجردة" هو قصر نظر وتجاهل جبال البيّنات العملية والتجريبية لضد هذا الأمر. تطوير البرمجيات هو تمرين خلّاق. إن لم تفهم الطريقة الخلّاقة، فإنك لا تحصل على معظم ما يمكن أن تقدمه الآلة الرئيسية التي تستخدمها لتنشئ برمجية-دماغك. تضيّع الطريقة السيئة دورات دماغك. ترفع الطريقة الجيدة فائدة دورات دماغك إلى أقصى فائدة.

34. 3 اكتب برامجاً للناس أولاً وللحواسيب ثانياً

برنامجك رقم ن. متاهة من الفوضى المستحيلة التخمين مع حِيَل شديدة الذكاء وتعليقات بعيدة عن الموضوع. قارن مع برنامجي.

برنامجي رقم ن. *جوهرة من الدقة الخوارزمية الحسابية، تعرض التوازن الأسمى بين كتابة الشفرة المتراصة والفعالة من جهة ووضوح تام التوثيق للأجيال القادمة من الجهة الأخرى. قارن مع* ***برنامجك.***

--ستان كيلي-بوتل

موضوع آخر يشغل هذا الكتاب هو التأكيد على قابلية قراءة الشفرة. التواصل مع الناس الآخرين هو الحافز خلف البحث عن الكأس المقدسة للشفرة ذاتية التوثيق.

[1] **إشارة مرجعية** لتفاصيل حول التكرار، انظر القسم 34. 8، "كرر، مجدداً، مرة بعد مرة،" لاحقاً في هذا الفصل.

لا يبالي الحاسوب إن كانت شفرتك مقروءة أم لا. إنه أفضل في قراءة تعليمات الآلة الثنائية منه في قراءة عبارات لغة عالية المستوى. فأنت تقوم بكتابة شفرة مقروءة لأنها تساعد الناس الآخرين على قراءة شفرتك. تمتلك قابلية القراءة تأثيراً إيجابياً على هذه الجوانب من البرنامج:

- قابلية الفهم
- قابلية المراجعة
- معدل الخطأ
- التصحيح
- قابلية التعديل
- زمن التطوير-نتيجة لكل ما سبق
- الجودة الخارجية-نتيجة لكل ما سبق

لا تأخذ الشفرة المقروءة مطلقاً وقتا أطول في كتابتها من الشفرة المربكة، على الأقل في الجري الطويل.[1] من السهل أن تتأكد أن شفرتك تعمل إن كنت تستطيع أن تقرأ ما كتبت بسهولة. هذا ينبغي أن يكون سبباً وجيهاً لتكتب شفرة مقروءة. لكن الشفرة تُقرأ أيضاً خلال المراجعات. وهي تُقرأ عندما تصلح أنت أو غيرك خطأ. وهي تُقرأ عندما تُعدَّل. وهي تقرأ عندما يحاول أحد ما أن يستخدم جزءاً من شفرتك في برنامج مشابه.

أن تجعل الشفرة مقروءة ليس جزءاً اختيارياً من عملية التطوير، وتفضيل أريحية الكتابة على أريحية القراءة اقتصاد خاطئ. ينبغي أن تبتغي السبيل لكتابة شفرة جيدة، والتي ستكتبها مرة واحدة، بدلاً من السبيل لقراءة شفرة سيئة، والتي ستقرؤها مجدداً ومجدداً.

"ماذا إن كنت أكتب شفرة لنفسي فقط؟ لمَ ينبغي أن أجعلها مقروءة؟" لأنه بعد أسبوع أو اثنين من الآن ستكون تعمل في برنامج آخر وتفكر، "هيي! لقد كتبت هذا الصف الأسبوع الماضي. فقط سأعود إلى شفرتي *المصححة* المُختبَرة القديمة وأوفر بعض الوقت." إن كانت الشفرة غير مقروءة، فحظاً موفقاً!

فكرة كتابة شفرة غير مقروءة لأنك الشخص الوحيد الذي يعمل على مشروع تهيئ سابقة خطيرة. اعتادت أمك أن تقول، "ماذا لو تجمّد وجهك في ذلك التعبير؟" واعتاد والدك أن يقول، " ستلعب كما تدربت." تؤثر العادات على كل ما تعمل؛ لا تستطيع أن تشغّلها أو تطفيها برغبتك، لذا كن متأكداً من أن ما تفعله هو شيء تريد أن يصبح عادة. يكتب المبرمج المحترف شفرة مقروءة، انتهى.

---

1 في السنين الأولى من البرمجة، كان يعتبر البرنامج كملكية خاصة للمبرمج. لا أحد سيفكر بقراءة برنامج زميل الدراسة بلا إذن أكثر مما يفكر باستلال رسالة حب وقراءتها. هذا ما كان البرنامج بالجوهر، رسالة حب من المبرمج إلى العتاد الصلب، مليئة بتفاصيل حميمة معروفة فقط بين الشريكين في العلاقة. وهكذا زُخرفت *البرامج بأسماء الدلع والاختصارات اللفظية الشائعة كثيراً بين العشاق الذين يعيشون في التجريد الهنيء الذي يَفترض أنهم هم الوجود الوحيد في العالم. هكذا مبرمجون غامضون لأولئك الذين هم خارج العلاقة. -مايكل ماركوتي*

أيضاً من الجيد إن وُجدت قطعة من الشفرة مُتنازع عليها أن تتعرف عليها أنها حصرياً لك في أي وقت. أتى دوغلاس كمر بتمييز مفيد بين البرامج الخاصة والعامة (كمر 1981): "البرامج الخاصة" هي برامج للاستخدام الشخصي للمبرمج. إنها لا تُقرأ من الآخرين. إنها لا تُعدّل من الآخرين. ولا يعرف الآخرون حتى بوجودها. إنها عادة بسيطة، وهي الاستثناء النادر. "البرامج العامة" هي برامج تُستخدم أو تُعدّل من أحد آخر غير الكاتب.

المعايير حول البرامج العامة والخاصة يمكن أن تختلف. البرامج الخاصة يمكن أن تكون مكتوبة بفوضوية ومليئة بالقصور بدون التأثير على أي كان ما عدا الكاتب. يتحتّم على البرامج العامة أن تكون مكتوبة بحذر أكثر: قصورها ينبغي أن يكون موثقاً، ينبغي أن تكون مقروءة، وينبغي أن تكون قابلة للتعديل. كن حذراً من تَطوُّر البرامج الخاصة إلى عامة، كما تفعل البرامج الخاصة عادة. إنك تحتاج أن تُحول البرنامج إلى برنامج عام قبل أن تدعه يذهب إلى الانتشار العام. جزء من جعل برنامج خاص عاماً هو جعله مقروءاً.

بيانات مثبتة

حتى وإن كنت تعتقد أنك الشخص الوحيد الذي سيقرأ شفرتك، في الواقع فرص جيدة أن أحداً آخر سيحتاج أن يعدّل شفرتك. وجدت دراسة واحدة أن 10 أجيال من مبرمجي الصيانة عملوا على برنامج متوسط قبل أن تُعاد كتابته (توماس 1984). يصرف مبرمجو الصيانة 50 إلى 60 بالمئة من وقتهم وهم يحاولون أن يفهموا الشفرة التي يقومون بصيانتها، ويقدّرون كل الوقت الذي تضعه في خدمة توثيقها (باريخ وزفيغينتوف 1983).

فحصت الفصول الأولى من هذا الكتاب التقنيات التي تساعدك في إنجاز سهولة القراءة: الصفوف والإجرائيات وأسماء المتحولات الجيدة، والتنسيق الحذِر، والإجرائيات الصغيرة، وإخفاء المنطق المعقد في توابع منطقية، وإسناد نتائج وسيطية إلى متحولات من أجل توضيح حسابات معقدة، وهلم جراً. ولا أي تطبيق مفرد من التقنيات يستطيع أن يصنع فرقاً بين برنامج مقروء وبرنامج غير مقروء، لكن تراكم عدة تحسينات صغيرة للقراءة سيكون واضحاً.

إن كنت تعتقد أنك لا تحتاج أن تجعل شفرتك مقروءة لأنْ لا أحد غيرك أبداً سينظر إليها، تأكد من أنك لا تخلط بين السبب والأثر.

34.4 برمج في لغتك، وليس بها

لا تقيّد تفكيرك البرمجي بالمفاهيم المدعومة تلقائياً من لغتك فقط. يفكر المبرمجون الأفضل بما يريدون فعله، وبعدها يجزمون كيف ينجزون أهدافهم بأدوات البرمجة تحت التَّصرُّف.

أعليك أن تستخدم إجرائية عضو في صف غير متناغمة مع تجريد الصف فقط لأن هذا أكثر راحة من استخدام أخرى تؤمن تماسكاً أكثر؟ ينبغي عليك أن تكتب شفرة بطريقة تحافظ على التجريد الممثل بواجهة الصف قدر الإمكان. لست مضطراً لاستخدام البيانات الشاملة أو *gotos* فقط لأن لغتك تدعمها. تستطيع أن تختار ألّا تستخدم هذه المقدرات البرمجية الخطيرة وبدلاً تستخدم أعراف برمجية لتُجمّل مناطق الضعف في اللغة. البرمجة باستخدام الطريق الأكثر وضوحاً يخلص إلى كونه برمجة باللغة وليس برمجة في اللغة؛ إنه المكافئ لدى المبرمج ل "إن انطلق زيد من الجسر، هل ستنطلق أنت من الجسر أيضاً؟" فكر بأهدافك التقنية، وبعدها قرر أفضل كيفية لإنجاز هذه الأهداف بالبرمجة في لغتك.

لا تدعم لغتك التأكيدات؟ اكتب إجرائية *assert()* الخاصة بك. قد لا تعمل بالضبط ك *assert()* القادمة مع اللغة، لكن لا تزال تستطيع تحقيق معظم فوائد *assert()* بكتابة إجرائيتك الخاصة. لا تدعم لغتك الأنماط التعدادية أو الثوابت المسماة؟ هذا جيد؛ تستطيع أن تُعرِّف تعداداياتك وثوابتك المسماة بالاستخدام المنضبط للمتحولات الشاملة المُزوَّدة بأعراف تسمية واضحة.

في الحالات فوق الحد، وخصوصاً في بيئات جديدة على البرمجة، قد تكون أدواتك بدائية جداً بحيث تضطر بشكل واضح إلى تغيير نهجك البرمجي المرغوب. في هكذا حالات، يجب عليك أن توازن بين رغبتك في البرمجة في اللغة والصعوبات العرضية التي تُخلق عندما تجعل اللغة نهجك المرغوب ثقيلاً جداً. لكن في هكذا حالات، ستستفيد أكثر حتى من أعراف البرمجة التي تساعدك في الابتعاد عن الميزات الأكثر خطورة لهذه البيئات. في حالات أكثر نمطية، الهوّة بين ما تريد أن تفعل وما ستدعمه أدواتك حالاً، سيتطلب منك أن تقدم فقط تنازلاً ثانوياً نسبياً للغتك.

34. 5 ركز انتباهك بمساعدة الأعراف

مجموعة من الأعراف هي إحدى الأدوات الفكرية المستخدمة لإدارة التعقيد.[1] تحدثت الفصول السابقة عن الأعراف الخاصة. يوضح هذا الفصل فوائد الأعراف بالعديد من الأمثلة.

العديد من تفاصيل البرمجة كيفية بطريقة ما. كم مسافة تضع قبل الحلقة؟ كيف تنسق التعليق؟ كيف ينبغي أن ترتب إجرائيات الصف؟ معظم الأسئلة المشابهة لهذه لها عدة أجوبة صحيحة. الطريقة المحددة التي يُجاب بها على أسئلة كهذه هو أقل أهمية من أن يُجاب بثبات في كل مرة. تنقذ الأعراف المبرمجين من مشكلة الإجابة على نفس السؤال-اتخاذ نفس القرارات الكيفية-مجدداً ومجدداً. في مشاريع فيها العديد من المبرمجين، استخدام الأعراف يمنع التصادم الناتج عندما يتخذ مبرمجون مختلفون القرارات الكيفية بشكل مختلف.

[1] **إشارة مرجعية** من أجل تحليل لقيمة الأعراف عندما تُطبَّق على تنسيق البرنامج، انظر "كم يستحق التنسيق الجيد؟" و "أهداف التنسيق الجيد" في القسم 31. 1.

يوصل العرف المعلومات الهامة بإيجاز. في أعراف التسمية، محرف واحد يمكن أن يميّز بين المتحولات المحلية والتابعة للصف والشاملة، الكتابة بحروف كبيرة يمكن بإيجاز أن يميّز بين الأنماط والثوابت المسماة والمتحولات. أعراف المسافات البادئة تستطيع بإيجاز أن تُظهر البنيان المنطقي للبرنامج. أعراف المحاذاة "الرصف" يمكن أن تحدد بإيجاز العبارات ذات الصلة.

تحمي الأعراف من المجازفات المعروفة. تستطيع أن تؤسس أعرافاً لتقصي استخدام التطبيقات المُجازِفة، أو لتقيّد هكذا تطبيقات إلى حالات الضرورة، أو لتعوّض عن هذه المجازفات المعروفة. تستطيع أن تقصي تطبيقاً خطيراً، مثالاً، بتحريم المتحولات الشاملة أو تحريم عدة عبارات على سطر واحد. تستطيع أن تعوض عن التطبيقات المجازفة بفرض أقواس حول التعابير المعقدة أو فرض أن تسند المؤشرات إلى NULL فور حذفها لتساعد منع تدلّي المؤشرات.

تضيف الأعراف قابلية التنبؤ لمهام المستوى المنخفض. أن يكون لديك طرقاً عرفية لتتعامل مع طلبات الذاكرة، ومعالجة الأخطاء، والدخل\الخرج، وواجهات الصفوف يضيف هيكلة ذات معنى لشفرتك ويجعلها أسهل للفهم على المبرمجين الآخرين-طالما يعرف المبرمج أعرافك. كما ذكر في فصل سابق، إحدى أهم فوائد إقصاء البيانات الشاملة هو أنك تقصي التفاعل المحتمل بين الصفوف والأنظمة الفرعية المختلفة. يعرف القارئ تقريباً ما يمكن أن تفعل البيانات المحلية أو بيانات الصف. لكن من الصعب أن يقول متى سيحطّم تغيير بيانات شاملة جزءاً ما من شفرة أربعة أنظمة فرعية. تزيد البيانات الشاملة من شك القارئ. مع أعراف جيدة، تستطيع أنت والقراء أن تضمن أشياء أكثر. كمية التفاصيل التي يتوجب أن تُستوعب ستُخفَّض، وهذا بدوره سيحسّن من فهم البرنامج.

تستطيع الأعراف أن تعوض عن مناطق الضعف في اللغة. في اللغات التي لا تدعم الثوابت المسماة (مثل **بايثون، بيرل وتدوين قشرة يونيكس** وهلم جراً)، يستطيع العرف أن يميّز بين المتحولات المُعدَّة لكلا القراءة والكتابة وتلك المعدّة لتحاكي ثوابت القراءة فقط. أعراف الاستخدام المنضبط للبيانات الشاملة والمؤشرات هي أمثلة أخرى عن التعويض عن مناطق الضعف في اللغة بوساطة الأعراف.

أحياناً في المشاريع الكبيرة يقوم المبرمجون باستخدام الأعراف بطريقة زائدة. إنهم يؤسسون معايير وتوجيهات كثيرة جداً بحيث يصبح تَذكُّرها عملَ وقتٍ كاملٍ. لكن في المشاريع الصغيرة يميل المبرمجون لاستخدام الأعراف بطريقة غير كافية، غير مدركين الفوائد الكاملة للأعراف المدروسة بذكاء. ينبغي أن تفهم قيمتها الحقيقية وتستفيد منها؛ ينبغي أن تستخدمها لتؤمن هيكلة في مناطق تلزم فيها الهيكلة.

34. 6 برمج بمفردات مجال المشكلة

طريقة محددة أخرى للتعامل مع التعقيد هي أن تتعامل مع المستوى الأعلى من التجريد. إحدى طرق التعامل مع المستوى الأعلى من التجريد هي أن تتعامل بمفردات المشكلة البرمجية وليس حلول العلم الحاسوبي.

لا ينبغي أن تكون شفرة المستوى الأعلى مليئة بتفاصيل عن الملفات والمكدسات والأرتال والمصفوفات والمحارف التي لم يجد أبويها أسماء أفضل لها مثل *i, j, k*. ينبغي أن تصف شفرة المستوى الأعلى المشكلة التي يتم حلها. ينبغي أن تكون مُحزَّمة بأسماء صفوف واستدعاءات توابع معبرة تحدد بالضبط ما يقوم به البرنامج، ولا تكون مُشوَّشة بتفاصيل عن فتح ملف "للقراءة فقط." لا ينبغي أن تحوي شفرة المستوى الأعلى كميات كبيرة من تعليقات تقول "*i* هو متحول يمثل فهرس السجل من ملف الموظفين هنا، وبعد قليل يُستخدم ليُفهرِس ملف حسابات الزبائن هناك."

ذلك تطبيق برمجي أخرق. في المستوى الأعلى من البرنامج، لست مضطراً لتعرف أن بيانات الموظفين تأتي كسجل أو أنها مُخزَّنة كملف. المعلومات في ذلك المستوى من التفصيل يجب أن تكون مخفية. في المستوى الأعلى، لا ينبغي أن تمتلك أي فكرة عن كيفية تخزين البيانات. ولا تحتاج أن تقرأ تعليقاً يشرح ماذا يعني *i* وأنه يُستخدم لغرضين. ينبغي أن ترى اسمي متحولين مختلفين للغرضين بدلاً، وينبغي أن تمتلك أسماء مميزة ك *employeeIndex* و *clientIndex*.

تقسيم برنامج إلى مستويات من التجريد

كما هو واضح، عليك أن تعمل بمفردات مستوى التحقيق في بعض المستويات، لكن تستطيع في البرنامج أن تعزل القسم الذي يعمل بمفردات مستوى التحقيق من البرنامج عن القسم الذي يعمل بمفردات مجال المشكلة. إن كنت تصمم برنامجاً، ضع في بالك مستويات التجريد الظاهرة في الشكل 34-1.

4 مفردات مجال المشكلة عالية المستوى
3 مفردات مجال المشكلة منخفضة المستوى
2 هيكليات التحقيق منخفضة المستوى
1 هيكليات وأدوات لغة البرمجة
0 عمليات نظام التشغيل وتعليمات الآلة

الشكل 34-1 يمكن تقسيم البرامج إلى مستويات من التجريد. سيتيح لك التصميم الجيد أن تصرف المزيد من وقتك بالتركيز على الطبقات العليا فقط وإهمال الطبقات السفلى.

المستوى 0: عمليات نظام التشغيل وتعليمات الآلة

إن كنت تعمل بلغة عالية المستوى، لا يجب عليك أن تقلق بشأن الطبقة السفلى-تهتم بها لغتك تلقائياً. إن كنت تعمل بلغة منخفضة المستوى، ينبغي أن تحاول إنشاء طبقات أعلى تزيد من راحتك لتعمل بها، حتى ولو كان معظم المبرمجين لا يفعلون ذلك.

المستوى 1: هيكليات وأدوات لغة البرمجة

هيكليات لغة البرمجة هي أنماط البيانات البدائية، وبنى التحكم، وهلم جراً. معظم اللغات الشائعة تزود مكتبات إضافية، ومداخل إلى استدعاءات نظام التشغيل، وهلم جراً. يأتي استخدام هذه الهيكليات والأدوات بشكل طبيعي إذ أنك لا تستطيع أن تبرمج بدونهما. العديد من المبرمجين لا يعملون فوق هذا المستوى من التجريد، ما يجعل حياتهم أصعب بكثير مما يلزم.

المستوى 2: هيكليات التحقيق منخفضة المستوى:

هيكليات التحقيق منخفضة المستوى هي هيكليات ذات مستوى أعلى بقليل من تلك المزودة من قبل اللغة نفسها. إنها تميل لتكون العمليات وأنماط البيانات التي تعلمت عنها في مناهج الكلية في الخوارزميات وأنماط البيانات: المكدسات، والأرتال، واللوائح المترابطة، والأشجار، والملفات المفهرسة، والملفات التراتبية، وخوارزميات الترتيب، وخوارزميات البحث، وهلم جراً. إن كان برنامجك بالكامل يتألف من شفرة مكتوبة بهذا المستوى، ستكون عائماً على الكثير جداً من التفاصيل لتفوز بالمعركة على التعقيد.

المستوى 3: مفردات مجال المشكلة منخفضة المستوى

في هذا المستوى، لديك الأساسيات التي تحتاج لتعمل بمفردات مجال المشكلة. إنها طبقة لاصقة بين هيكليات علوم الحاسوب من تحت وشفرة مجال المشكلة عالية المستوى من فوق. لتكتب شفرة في هذا المستوى، تحتاج أن تفكر بمفردات مجال المشكلة وتنشئ حجارة البناء التي تستطيع أن تستخدمها لتتعامل مع المشكلة التي يحلها البرنامج. في الكثير من التطبيقات، ستكون هذه طبقة أهداف العمل أو طبقة الخدمات. تقدم الصفوف في هذا المستوى المفردات وحجارة البناء. قد تكون الصفوف بدائية جداً على الاستخدام في حل المشكلة مباشرة في هذا المستوى، لكنها تقدم منصة تستطيع صفوف الطبقة التي فوقها أن تستخدمها لتبني حلاً للمشكلة.

المستوى 4: مفردات مجال المشكلة عالية المستوى

يؤمن هذا المستوى القوة التجريدية للتعامل مع المشكلة بمفرداتها. ينبغي أن تكون شفرتك في هذا المستوى مقروءة بطريقة أو بأخرى لشخص آخر ليس خبيراً بعلم الحاسوب، ربما حتى زبونك غير المختص تقنياً. لن تعتمد الشفرة في هذا المستوى كثيراً على المميزات الخاصة بلغة برمجتك لأنك ستكون قد بنيت طقمك الخاص من الأدوات لتتعامل مع المشكلة. بناء على ذلك، تعتمد شفرتك في هذا المستوى على الأدوات التي بنيتها لنفسك في المستوى 3 أكثر من مقدرات اللغة التي تستخدمها.

ينبغي أن تكون تفاصيل التحقيق مخفية تحت طبقتين من هذه، في طبقة هيكليات علم الحاسوب (المستوى 2)، بحيث لا تؤثر التغييرات في العتاد أو نظام التشغيل على هذه الطبقة على الإطلاق. جسّد نظرة المستخدم للعالم في البرنامج في هذا المستوى لأنه عندما يتغير البرنامج، سيتغير حسب نظرة المستخدم. ينبغي أن تؤثر التغيرات في مجال المشكلة على هذه الطبقة كثيراً، لكن ينبغي أن يكون من السهل ملائمتها عبر البرمجة بكتل البناء التابعة لمجال المشكلة المعرفة في الطبقة التي تحتها.

بالإضافة إلى هذه الطبقات المفاهيمية، يجد العديد من المبرمجين أن تُجزِّئ البرنامج إلى "طبقات" أخرى تختصر الطبقات الموصوفة هنا مفيداً. مثالاً، الهيكلة ثلاثية الطبقات النمطية تختصر المستويات الموصوفة هنا وتؤمن أدوات إضافية لجعل التصميم والشفرة قابلين للإدارة فكرياً.

تقنيات منخفضة المستوى للعمل في مجال المشكلة

حتى بدون نهج هيكلي كامل للتعامل بمفردات مجال المشكلة، تستطيع أن تستخدم العديد من التقنيات الموجودة في هذا الكتاب لتعمل بمفردات مشكلة العالم الحقيقي بدلاً من الحل المتعلق بعلم الحاسوب:

- استخدم صفوفاً لتحقق هيكليات ذات معنى في مصطلحات مجال المشكلة.
- خبئ المعلومات المتعلقة بأنماط بيانات المستوى المنخفض وتفاصيل تحقيقها.
- استخدم الثوابت المسماة لتوثق معاني السلاسل النصية والمحارف الرقمية.
- عيّن متحولات وسيطة لتوثق نتائج الحسابات الوسيطة.
- استخدم توابعاً منطقية لتوضح اختبارات المنطق المعقدة.

34. 7 انتبه من الصخور الساقطة

البرمجة ليست فناً بشكل كامل وليست علماً بشكل كامل. كما تُطبَّق عادة، إنها "حرفة" في مكان ما بين الفن والعلم. بشكلها الأفضل، هي فرع من فروع الهندسة التي نشأت من الاندماج التعاوني بين الفن والعلم (ماك كونيل 2004). سواء أكانت فن أو علم أو حرفة أو هندسة، فإنها لا تزال تأخذ الكثير من المحاكمة الفردية لتخلق منتجاً برمجياً شغّالاً. وجزء من امتلاك المحاكمة الجيدة في برمجة الحاسوب هو الكون حساساً لصفٍّ

عريض من إشارات التحذير، والعلامات المخادعة المتعلقة بالمشاكل في برنامجك. تنذرك إشارات التحذير في البرمجة إلى احتمالية المشاكل، لكنها عادة ليست صارخة بمقدار إشارة طريق تقول "انتبه من الصخور الساقطة."

عندما تقول أنت أو شخص آخر "هذه حقاً شفرة مخادعة،" فإن هذا إشارة تحذير، عادةً، لشفرة رديئة. "شفرة مخادعة" هي عبارة اصطلاحية ل "شفرة سيئة." إن كنت تعتقد أنَّ شفرةً مخادعةٌ، فكر بإعادة كتابتها بحيث لا تكون كذلك.

صف يمتلك أخطاء أكثر من المعدل هي إشارة تحذير. تميل قلة من الصفوف الميالة للخطأ لتكون هي الجزء الأغلى من برنامج. إن كان لديك صف قد امتلك أخطاء أكثر من المعدل، فقد يتابع بامتلاك أخطاء أكثر من المعدل. فكر بإعادة كتابته.

إن كانت البرمجة علماً، ستقتضي كل إشارة تحذير فعل تصحيح معرّفاً جيداً ومحدداً. لأن البرمجة لا تزال حرفة، على كل، تدل إشارة التحذير إلى قضية ينبغي أن تُؤخذ بعين الاعتبار فقط. لن تستطيع حتماً أن تعيد كتابة شفرة مخادعة أو أن تحسّن صفاً ميالاً للخطأ.

فقط كما يحذرك العدد غير الطبيعي للعيوب في صف من أن الصف يمتلك جودة منخفضة، العدد غير الطبيعي للعيوب في برنامج يقتضي أن طريقتك مختلّة. الطريقة الجيدة لن تسمح للشفرة الميالة للخطأ بأن تَّطور. ستتضمن الفحوصات والموازنات للهيكلة متبوعة بمراجعات الهيكلة، والتصميم متبوعاً بمراجعات التصميم، والشفرة متبوعة بمراجعات الشفرة. مع مرور الوقت والشفرة تتهيأ للاختبار، معظم الأخطاء ستُستأصل. يتطلب الأداء الاستثنائي حتماً عملاً بذكاء بالإضافة إلى عمل بجدّ. كثرة *التصحيحات* في مشروع هي إشارة تحذير تقتضي أن الناس لا يعملون بذكاء. كتابة الكثير من الشفرة في يوم ومن ثم قضاء أسبوعين في *تصحيحها* ليس عملاً بذكاء.

تستطيع استخدام علم قياسات التصميم كنوع آخر من إشارات التحذير. معظم قياسات التصميم استدلالية وتعطي مؤشراً لجودة التصميم. حقيقة كون صف يحتوي أكثر من سبعة أعضاء لا تعني بالضرورة أنه صُمم بشكل رديء، بل هي تحذير أن الصف معقد. بشكل مشابه، أكثر من حوالي 10 نقاط اتخاذ قرار في إجرائية، وأكثر من ثلاث مستويات من التعشيش المنطقي، وعدد غير طبيعي من المتحولات، واقتران شديد بصفوف أخرى، وتماسك ضعيف ضمن الإجرائية أو الصف هي علامات أي منها ينبغي أن يرفع علم التحذير. ولا واحدة من هذه العلامات تعني أن الصف مُصمم بشكل رديء، بل حضور أي واحدة منها ينبغي أن يدعوك لتنظر إلى هذا الصف بنظرة الشك.

ينبغي لأي إشارة تحذير أن تسبب في نفسك شكاً حول جودة برنامجك. كما قال شارلز ساوندرز بيريس، "الشك هو حالة الهم والانزعاج والتي منها نكافح لنحرر أنفسنا ونمرق إلى حالة الإيمان." عامل إشارة التحذير ك "إثارة للشك" والتي تدعوك لتبحث عن حالة إيمانية أكثر إرضاء من الإيمان.

إن وجدت نفسك تعمل على شفرة متكررة أو تصنع تعديلات متماثلة في عدة مناطق، عليك أن تشعر ب "هم وانزعاج،" وتكون مرتاباً حول كون التحكم متمركزاً بشكل ملائم في صفوف أو إجرائيات أم لا. إن وجدت أنه من الصعب أن تنشئ سقالة لاختبار حالات لأنك لا تستطيع استخدام صف مفرد بسهولة، ينبغي أن تشعر "بإثارة للشك" وتسأل إن كان الصف مقترناً بصفوف أخرى بإحكام شديد. إن لم تستطع إعادة استخدام شفرة في برامج أخرى لأن بعض الصفوف معتمدة على بعضها كثيراً، هذه إشارة تحذير أخرى بأن هذه الصفوف مقترنة بإحكام شديد.

عندما تغوص في برنامج، انتبه إلى إشارات التحذير التي تدل على أن جزء من تصميم البرنامج لم يكن مُعرَّفاً بشكل جيد كفاية حتى تُكتب الشفرة. الصعوبات في كتابة تعليقات، وتسمية متحولات، وتحليل المشكلة إلى صفوف متماسكة لها واجهات واضحة كل ذلك يدل على أنك تحتاج أن تفكر بجد أكثر في التصميم قبل كتابة الشفرة. الأسماء الواهنة والصعوبة في وصف أقسام من الشفرة بتعليقات موجزة هما علامات أخرى للمشاكل. عندما يكون التصميم واضحاً في عقلك، تأتي تفاصيل المستوى المنخفض بسهولة.

كن حساساً لإشارات تدل على أن برنامجك صعب الفهم. إي مشقة هي دليل. إن كان صعباً عليك، سيكون حتى أصعب على المبرمجين اللاحقين. سيقدّرون الجهد الزائد الذي تبذله لتحسّنه. إن كنت تكتشف الشفرة بدلاً من أن تقرأها، فهي معقدة جداً. إن كانت صعبة، فهذا أمر خاطئ. اجعلها أبسط.

بيانات مثبتة

إن كنت تريد أن تأخذ كل فائدة إشارات التحذير، برمج بطريقة وكأنك تنشئ تحذيراتك الخاصة. هذا مفيد لأنه حتى بعد أن تعرف ما هي الإشارات، فإنه أمر سهل بشكل مدهش أن تغفل عنها. قاد غلينفورد مايرز دراسة عن تصحيح العيوب وجد بها أن السبب المفرد الأكثر شيوعاً لعدم إيجاد الأخطاء كان ببساطة الغفلة عنها. كانت الأخطاء ظاهرة في خرج الاختبار لكنها غير ملحوظة (مايرز 1978 b).

اجعل الغفلة عن المشاكل في برنامجك أمراً صعباً. أحد الأمثلة هو تعيين المؤشرات إلى لا شيء بعد أن تحررها بحيث ستسبب لك مشاكل بشعة إن استخدمت أحدها بشكل خاطئ. قد يؤشر المؤشر المحرر إلى موقع ذاكرة صالح حتى بعد أن يُحرَّر. تعيينه إلى لا شيء يضمن أنه يؤشر إلى موقع غير صالح، ما يجعل الغفلة عن الخطأ أصعب.

تحذيرات المترجم هي إشارات تحذير حرْفية والتي غالباً ما تُغفل. إن كان برنامجك يولد تحذيرات أو أخطاء، أصلحه بحيث لا يولّد. ليس لديك فرصة كبيرة لملاحظة إشارات تحذير دقيقة عندما تتجاهل تلك التي فيها "WARNING" مطبوعة بشكل مباشر فيها.

لماذا لفْت الانتباه إلى إشارات التحذير الفكرية هام بشكل خاص في تطوير البرمجيات؟ جودة التفكير الذي تسير به في برنامج يحدد بشكل ضخم جودة البرنامج، لذلك لفت الانتباه إلى تحذيرات حول جودة التفكير تؤثر بشكل مباشر على المنتج النهائي.

34. 8 كرر، مجدداً، مرة بعد مرة

التكرار مناسب للعديد من نشاطات تطوير البرمجيات. خلال توصيفك الأولي لنظام، تعمل مع المستخدم على إصدارات متعددة من المتطلبات حتى تتأكد أنكم اتفقتم عليها. هذه عملية تكرارية. عندما تنشئ مرونة في طريقتك ببناء وتسليم نظام بزيادات متعددة، فإن هذا عملية تكرارية. إن كنت تستخدم النماذج الأولية لتطور حلول بديلة متعددة بسرعة ورخص قبل أن تشكل ببراعة المنتج النهائي ببراعة، فإن هذا شكل آخر من التكرار. التكرار في المتطلبات ربما بأهمية أي جانب آخر من جوانب عملية تطوير البرمجيات. تفشل المشاريع لأنها ترهن نفسها لحل قبل أن تكتشف البدائل. يؤمن التكرار طريقة لتتعلم عن منتج قبل أن تبنيه.

كما أشار الفصل 28، "إدارة البناء،" يمكن أن تتغير تخمينات جداول المواعيد خلال التخطيط الأوليّ للمشروع بشكل كبير وفق تقنية التخمين التي تستخدمها. استخدام نهج تكراري للتخمين ينتج تخميناً أكثر دقة من الاعتماد على تقنية مفردة.

تصميم البرمجية هو عمل استكشافي و، ككل العمليات الاستكشافية، هو موضوع للتنقيح والتحسين التكراريين. تميل البرمجيات لتكون موضوعاً لجعلها صالحة وليس لإثباتها، والذي يعني أنها تُختبر وتُطوَّر بشكل تكراري حتى تجيب على الأسئلة بشكل صحيح. محاولات كلا التصميمين عالي المستوى ومنخفض المستوى ينبغي أن تُعاد. قد تُنتج محاولة أولى حلاً يعمل، لكنها غير مرجحة لإنتاج أفضل حل. اتخاذ نهج مكررة ومختلقة متعددة تعطي بصيرة بالمشكلة لا تكون مرجحة الظهور بنهج مفرد.

تَظهر فكرة التكرار مجدداً في معايرة الشفرة. حالما تكون البرمجية جاهزة للعمل، تستطيع أن تعيد كتابة أجزاء صغيرة منها لتحسّن أداء النظام الكلي بشكل عظيم. العديد من المحاولات في المعايرة، على كل، تؤذي الشفرة أكثر مما تساعدها. إنها ليست عملية معتمدة على الحدس، وبعض التقنيات التي تبدو بشكل مرجح أنها ستجعل النظام أصغر وأسرع تقوم بالفعل بجعله أكبر وأبطأ. عدم التأكد بشأن تأثير أي تقنية معايرة يخلق حاجة لمعايرة،

وقياس، ومعايرة مجدداً. إن كان عنق زجاجة أمرا حرجا في أداء النظام، تستطيع أن تعاير الشفرة عدة مرار، وعدة من محاولاتك الأخيرة قد تكون أكثر نجاحاً من أولاها.

تقطع المراجعات خط عملية التطوير بشكل معامد، حاشرة تكرارات في أي مرحلة تكون مسؤولة عنها. الغرض من المراجعة هو اختبار جودة العمل في نقطة محددة. إن فشل المنتج في المراجعة، فإنه يُعاد لإعادة العمل. إن نجح، فإنه لا يحتاج تكراراً بعد ذلك.

أحد تعاريف الهندسة هو أن تعمل مقابل عشرة سنتات ما يستطيع أي شخص فعله مقابل دولار. التكرار في المراحل الأخيرة هو أن تعمل بدولارين ما يستطيع أي شخص فعله مقابل دولار واحد. اقترح فريد بروكس "ابنِ شيئاً لترميه بعيداً؛ ستفعل ذلك، بأي حال" (بروكس 1995). الحيلة في هندسة البرمجيات هي أن تبني بالسرعة والرخص الممكنين الأجزاء القابلة للانفصال، والتي هي غاية التكرار من المراحل المبكرة.

34. 9 افصل بقوة بين البرمجيات والدين

يَظهر الدين في تطوير البرمجيات بعدة تجسيدات-كالالتزام الجازم بمنهجية تصميم مفردة، أو كالإيمان الثابت بتنسيق محدد أو بأسلوب كتابة تعليقات، أو كالتجنب المتعصب للبيانات الشاملة. مهما تكن الحالة، إنه غير مناسب بكل الأوقات.

عرّافو البرمجيات

لسوء الحظ، الطريقة المتعصبة مفروضة من الأعلى من قبل بضعة من أشهر البشر في المهنة.[1] إنه لأمر مهم أن تنشر التجديد كي يستطيع الممارسون أن يجربوا المنهجيات الجديدة الواعدة. يجب أن تُجرَّب المنهجيات قبل التمكن من إثباتها أو نفيها بشكل كامل. نثر نتائج البحث إلى الممارسين يُدعى "نقل التقنية" وهو مهم لدفع حالة تطبيق التطوير البرمجي. هنالك فرق، على كل، بين نثر منهجية جديدة وبيع زيت الأفعى الخاص بالبرمجيات. فكرة نقل التقنية خُدمت بشكل رديء من قبل باعة المنهجيات العقائدية المتجولين الذين يحاولون أن يقنعوك أن فطائر البقر عالية التقنية، ذات الحجم الواحد والتي تناسب الجميع الخاصة بهم ستحل كل مشاكلك. انسَ كل شيء تعلمته مسبقاً لأن هذه المنهجية الجديدة عظيمة جداً وستحسن إنتاجيتك 100 بالمئة في كل شيء!

بدلاً من الانزلاق وراء آخر موضة للأعاجيب، استخدم خليطاً من المنهجيات. حاول أن تكتشف المنهجيات المثيرة والأحدث، لكن ادخر تلك القديمة والموثوقة.

[1] **إشارة مرجعية** لتفاصيل حول التعامل مع الدين البرمجي كمدير، انظر "قضايا عقائدية" في القسم 28. 5.

الاصطفائية

الاعتقاد الأعمى بطريقة واحدة يعيق الانتقائية التي تحتاجها إن كنت تريد أن تجد الحلول الأكثر فعالية للمشاكل البرمجية. [1]إن كان تطوير البرمجيات عملية حتمية وخاضعة للخوارزميات، فإنك تستطيع أن تتبع منهجية صلبة في حلّك. لكن تطوير البرمجيات ليس عملية حتمية، بل استكشافية، ما يعني أن العمليات الصلبة تكون غير مناسبة ولديها أمل قليل بالنجاح. في التصميم، مثالاً، أحياناً التحليل من الأعلى إلى الأسفل يعمل بشكل جيد. أحياناً النهج غرضي التوجه، أو التركيب من الأسفل إلى الأعلى، أو نهج هيكلة البيانات يعمل أفضل. عليك أن تكون راغباً بتجريب نُهج متعددة، وأنت تعلم أن بعضها سيخفق وبعضها سينجح لكن لا تعلم أي منها سينجح حتى تجربها. عليك أن تكون اصطفائي.

الالتصاق بمنهجية مفردة مؤذ أيضاً في أنه يجعلك تجبر المشكلة على موافقة الحلّ. إن اتخذت قراراً بشأن طريقة الحل قبل أن تفهم المشكلة بشكل كامل، فإنك تتصرف بشكل غير ناضج. قيّد بشكل زائد مجموعة الحلول الممكنة، وقد تلغي الحل الأكثر فعالية.

ستكون غير مرتاحاً مع أي منهجية في البداية، والنصيحة التي تقول أنه عليك تجنب الدِّين في البرمجة لا تعني اقتراحاً بأنه ينبغي أن تتوقف عن استخدام طريقة جديدة حالما تواجه مشكلة طفيفة في حل مشكلة بها. أعط الطريقة الجديدة مصافحة عادلة، وأعط الطرق القديمة مصافحاتهم العادلة أيضاً.

الاصطفائية[2] هي نمط مفيد لتتعامل بوعي مع التقنيات المُقدَّمة في هذا الكتاب بمقدار ما هو مفيد في التعامل مع التقنيات الموصوفة في مصادر أخرى. [3]تحتوي النقاشات حول مواضيع متعددة مُقدَّمة هنا نُهجاً متقدمة اختيارية والتي لا يمكنك أن تستخدمها في نفس الوقت. عليك أن تختار واحداً أو آخر لكل مشكلة محددة. عليك أن تعامل التقنيات كأدوات في صندوق العدة وأن تستخدم محاكمتك الخاصة لتختار الأداة الأفضل للعمل.

---

1 

[2] **إشارة مرجعية** لوصف مفصل أكثر لاستعارة صندوق الأدوات، انظر "تطبيق التقنيات البرمجية: صندوق الأدوات الفكري" في القسم 2. 3.

[3] **إشارة مرجعية** للمزيد عن الفرق بين النهجين الخوارزمي والاستكشافي، انظر القسم 2. 2، "كيف تستخدم الاستعارات البرمجية." لمعلومات عن الاصطفائية في التصميم، انظر "كرر" في القسم 5. 4.

في معظم الأوقات، اختيار الأداة لا يهم كثيراً جداً. تستطيع أن تستخدم مفتاح رنش مغلق، أو كماشة لقط، أو مفتاح رنش هلالي. في بعض الحالات، على كل، اختيار الأداة يهم كثيراً، لذا ينبغي دائماً أن تجعل اختيارك حذراً. الهندسة هي جزئياً فرع المعرفة المتعلق بصنع مقايضات بين التقنيات المتنافسة. لا تستطيع أن تصنع مقايضة إن كنت تقرن خيارك بأداة مفردة بطريقة غير ناضجة.

استعارة صندوق العدة مفيدة لأنها تجعل الفكرة المجردة للاصطفائية مشهودة. افترض أنك متعهد عام وزميلك سيمون البسيط يستخدم دائماً كماشة لقط. افترض أنه يرفض أن يستخدم مفتاح الرنش المغلق أو الهلالي. قد تفكر أنه شاذ لأنه لا يستخدم كل الأدوات التي بين يديه. نفس الأمر صحيح في تطوير البرمجيات. في مستوى عال، لديك طرق تصميم اختيارية. في مستوى أكثر تفصيلا، تستطيع أن تختار واحداً من عدة أنماط بيانات لتمثل أي تصميم معطى. في مستوى أكثر تفصيلا بعد، تستطيع اختيار مخططات مختلفة متعددة للتنسيق والتعليق على الشفرة، وتسمية المتحولات، وتعريف واجهات الصفوف، وتمرير وسطاء الإجرائيات.

الموقف الجازم يتعارض مع نهج صندوق العدة الاصطفائي لبناء البرمجيات. إنه غير متوافق مع النمط اللازم لبناء برمجيات عالية الجودة.

التجريب

للاصطفائية علاقة وثيقة بالتجريب. إنك تحتاج أن تجرب خلال سير عملية التطوير، لكن الصلابة المتعصبة تعيق الاندفاع. لتجرب بشكل فعال، يتوجب عليك أن تكون راغباً بتغيير معتقداتك بالاعتماد على نتائج التجربة. إن لم تكن راغباً، فإن التجريب هو مضيعة غير مبررة للوقت.

العديد من النُهج غير المرنة في تطوير البرمجيات تعتمد على الخوف من ارتكاب الأخطاء. المحاولة الشاملة لتجنب الأخطاء هي الخطأ الأكبر بين الكل. التصميم هو عملية التخطيط بحذر لأخطاء صغيرة لتجنب ارتكاب أخطاء أكبر. التجريب في تطوير البرمجيات هو عملية وضع اختبارات بحيث تتعلم أن كان نهج ما يفشل أو ينجح-التجربة بحد ذاتها هي نجاح طالما أنها تحل القضية.

التجريب مناسب بعدة مستويات بقدر ما هي مناسبة الاصطفائية. في كل مستوى فيه تكون جاهزاً لاتخاذ اختيار اصطفائي، ربما تستطيع أن تأتي بتجربة منسجمة لتحدد أي نهج يعمل بالطريقة الأفضل. في مستوى التصميم الهيكلي، قد تتألف التجربة من رسم هيكليات البرمجية باستخدام ثلاثة نُهج تصميمية مختلفة. في مستوى التصميم التفصيلي، قد تتألف التجربة من اتباع مقتضيات الهيكلة في المستوى الأعلى باستخدام ثلاثة نهج تصميمة منخفضة المستوى مختلفة. في مستوى لغة البرمجة، قد تتألف التجربة من كتابة برنامج اختباري قصير لتشغّل وظيفة جزء من اللغة لست أليفاً بها بشكل كامل. قد تتألف التجربة من معايرة قطعة من الشفرة

واختبار أدائها للتأكيد على أنها فعلاً أصغر أو أسرع. في مستوى عملية تطوير البرمجيات الكلية، قد تتألف التجربة من جمع بيانات الجودة والإنتاجية بحيث تستطيع أن ترى إن كانت التفحصات تكتشف أخطاء أكثر من العبورات.

الغاية هي أنه عليك أن تحتفظ بفكر منفتح على كل جوانب تطوير البرمجيات. عليك أن تكون تقنيا بشأن طريقتك بالإضافة إلى منتجك. التجريب ذو الفكر المنفتح والالتصاق الديني بنهج معرف مسبقاً لا يجتمعان.

نقاط مفتاحية

- أحد الأهداف الرئيسة في البرمجة هو إدارة التعقيد.
- طريقة البرمجة تؤثر بشكل ملحوظ بالمنتج النهائي.
- البرمجة في فريق هو تدريب في التواصل مع الناس أكثر من التواصل مع الحاسوب. البرمجة الفردية هي تدريب في التواصل مع النفس أكثر من التواصل مع الحاسوب.
- عندما يُساء استخدامه، يمكن للعرف البرمجي أن يكون علاجاً أسوأ من المرض، وباستخدامه بحكمة، يضيف العرف بنياناً قيّماً لبيئة التطوير ويساعد في إدارة التعقيد والتواصل.
- البرمجة بمفردات المشكلة بدلاً من الحل يساعد في إدارة التعقيد.
- لفت الانتباه إلى إشارات التحذير الفكرية مثل "إثارة الشك" أمر مهم في البرمجة بشكل خاص لأن البرمجة هي تقريباً نشاط عقلي على نحو محض.
- كلما كررت أكثر في أي نشاط من نشاطات التطوير، كلما كان ناتج ذلك النشاط أفضل.
- المنهجيات الجازمة وتطوير البرمجيات عالي الجودة لا يجتمعان. املأ صندوق أدواتك الفكري بالبدائل البرمجية، وحسّن خبرتك في اختيار الأداة الصحيحة للعمل.

الفصل 35

أين تجد معلومات إضافية

المحتويات[1]

مواضيع ذات صلة

- مصادر الوب: www.cc2e.com

إن كنت تقرأ إلى هذا البعد، فإنك تعلم مسبقاً أن الكثير قد كُتبَ عن تطبيقات تطوير البرمجيات الفعالة. الكثير من المعلومات متاح أكثر مما يدرك معظم الناس. ارتكب البشر مسبقاً الأخطاء التي تصنعها الآن، وما لم تكن شرهاً للعقاب، ستفضل قراءة كتبهم وتجنب أخطائهم على ابتكار إصدارات جديدة لمشاكل قديمة.

لأن هذا الكتاب يصف مئات الكتب الأخرى والمقالات الحاوية على معلومات عن تطوير البرمجيات، فإنه من الصعب أن تعرف بمَ تبدأ قراءة. مكتبة تطوير البرمجيات مكونة من عدة أنواع من المعلومات. الكتب عن جوهر البرمجة توضح المفاهيم الأساسية للبرمجة الفعالة. كتب ذات صلة توضح سياقات تقنية وإدارية وفكرية أوسع تسير البرمجة فيها. والمراجع المفصلة عن اللغات، وأنظمة التشغيل، والبيئات، والعتاد تحوي معلومات مفيدة لمشاريع خاصة.

الكتب في الفئة الأخيرة لها دورة حياة بحوالي مشروع واحد بالعموم؛[2] هي مؤقتة أكثر من ذلك أو أقل ولم تُناقَش هنا. بالنسبة للأنواع الأخرى من الكتب، من المفيد أن يكون لديك مجموعة جوهرية تناقش كل نشاطات تطوير البرمجيات الرئيسة بعمق: كتب عن المتطلبات والتصميم والبناء والإدارة والاختبار وهلم جراً. توضح

[1] cc2e.com/3560
[2] cc2e.com/3581

الأقسام التالية بعمق مصادر عن البناء وبعدها تؤمن نظرة عامة على المواد المتاحة في مناطق المعرفة البرمجية الأخرى. يغلف القسم 35. 4 هذه المصادر في حزمة أنيقة عن طريق تعريف برنامج قراءة لمطور البرمجيات.

35. 1 معلومات عن البناء البرمجي

كتبت هذا الكتاب بالأصل لأنني لم أستطع إيجاد نقاش عميق عن البناء البرمجي.[1] في سنوات منذ أن نشرت النسخة الأولى، عدة كتب جيدة ظهرت.

"درر البرمجة" Pragmatic Programmer (Hunt and Thomas 2000) يركز على النشاطات الأقرب صلةً بكتابة الشفرة، متضمناً الاختبار و*التصحيح* واستخدام التأكيدات وهلم جراً. إنه لا يغوص عميقاً غي الشفرة نفسها لكنه يحتوي مبادئ عديدة متعلقة بإنشاء شفرة جيدة.

Jon Bentley's Programming Pearls, 2d ed. (Bentley 2000) يناقش فن وعلم التصميم البرمجي بتواضع. الكتاب مُنظَّم كمجموعة من المقالات المكتوبة بشكل جيد جداً والمعبّرة عن كمية كبيرة من البصيرة في تقنيات البناء الفعال بالإضافة إلى تعصب أصيل إلى البناء البرمجي. أنا أستخدم شيئاً مما تعلمته من مقالات بنتلي تقريباً كل يوم أبرمج فيه.

" شرح البرمجة المتطرفة:[2] احتضان التغيير" Kent Beck's Extreme Programming Explained: Embrace Change(Beck 2000) يعرف نهجاً لتطوير البرمجيات متمركزاً على البناء. كما يوضح القسم 3. 1 ("أهمية المتطلبات") ("Importance of Prerequisites")، تأكيدات الكتاب حول اقتصاديات **البرمجة الزائدة** ليست مستخرجة من أبحاث صناعية، لكن العديد من نصائحه مفيدة خلال البناء سواء أكان فريقك يستخدم **البرمجة الزائدة** أو نهجاً آخر.

كتاب أكثر تخصص هو كتاب ستيف ميغوير " كتابة الشفرة الصلبة - تقنيات مايكروسوفت لتطوير برمجيات سي خالية من الأخطاء " Writing Solid Code – Microsoft's Techniques for Developing Bug-Free C Software(Maguire 1993)، إنه يركز على تطبيقات البناء لأجل تطبيقات برمجية ذات جودة تجارية، معظمها معتمدة على خبرة الكاتب من عمله على تطبيقات **مايكروسوفت أوفيس**. إنه يركز على تقنيات قابلة للتطبيق في **سي**. إنه بشكل كبير غافل عن قضايا البرمجة غرضية التوجه، لكن معظم المواضيع التي يعالجها ذات صلة بأية بيئة.

[1] cc2e.com/3588

[2] **إشارة مرجعية** للمزيد عن اقتصاديات **البرمجة الزائدة** والبرمجة الرشيقة، انظر cc2e.com/3545.

كتاب آخر أكثر تخصص هو "ممارسة البرمجة" The Practice of Programming, by Brian Kernighan and Rob Pike (Kernighan and Pike 1999). يركز هذا الكتاب على الجوانب التطبيقية الجوهرية للبرمجة، ويغلق الفجوة بين المعرفة الأكاديمية بعلوم الحاسوب والدروس العملية. إنه يتضمن نقاشات عن أسلوب البرمجة، والتصميم، و*التصحيح*، والاختبار. إنه يفترض إلفة ل **سي****سي** ++.

على الرغم من أنه لم يعد يًطبع، ومن الصعب إيجاده،[1] "المبرمجون في العمل" Programmers at Work, by Susan Lammers (1986)، فإنه يستحق البحث. إنه يحوي مقابلات مع مبرمجي المستوى الراقي الصناعي. تكشف المقابلات عن شخصياتهم، وعادات عملهم، وفلسفات البرمجة. الأشخاص اللامعون الذين تمت مقابلتهم منهم بل غيتس (مؤسس **مايكروسوفت**)، وجون وارنوك (مؤسس **أدوبي**)، وآندي هيرتزفيلد (مطور رئيس لنظام تشغيل **ماكنتوش**)، وبتلر لامبسون (كبير المهندسين في **ديك**، حالياً في **مايكروسوفت**)، واين راتليف (مخترع **دي بيز**)، دان بريكلن (مخترع **فيزي كالك**)، ودزينة من الآخرين.

35. 2 مواضيع وراء البناء

وراء الكتب الجوهرية الموصوفة في القسم السابق، هاهنا بعض الكتب التي تجوب بعيداً جداً عن موضوع البناء البرمجي.

مواد النظرة العامة

تقدم الكتب التالية نظرات عامة على تطوير البرمجيات من نقاط أفضلية متنوعة:[2]

"حقائق الصفوف ومغريات هندسة البرمجيات" Robert L. Glass's *Facts and Fallacies of Software Engineering* (2003)، يقدم مقدمة سهلة القراءة عن الحكمة العرفية لافعل ولا تفعل في تطوير البرمجيات. الكتاب مدروس بشكل جيد ويؤمن وصلات عديدة إلى مصادر إضافية.

"تطوير البرمجيات المهني" *Professional Sofware Development* (2004) الخاص بي، يستعرض مجال تطوير البرمجيات كما يُطبَّق الآن وكما من الممكن أن يكون إن طُبِّق بشكل متكرر بالطريقة الأفضل.

" دليل للهيكل الهندسي للمعرفة " The *Swebok*: *Guide to the Software Engineering Body of Knowledge* (Abran 2001) يقدم تحليلاً مفصلاً لكمية كبيرة من المعارف المتعلقة بهندسة البرمجيات. لقد

---

[1] cc2e.com/3549

[2] cc2e.com/3595

غاص هذا الكتاب إلى حقائق صغيرة في منطقة البناء البرمجي. يظهر الكتاب فقط كم هي كثيرة جداً المعلومات الموجودة في هذا المجال.

" علم نفس برمجة الحاسب" Gerald Weinberg's *The Psychology of Computer Programming*(Weinberg 1998) إنه مزدحم بالطرف الجذابة عن البرمجة. إنه يطوف بعيداً لأنه كُتب في وقت كان يعتبر فيه أن أي شيء مرتبط بالبرمجيات هو عن البرمجة. النصيحة في المراجعة الأصلية للكتاب في **مراجعات الحوسبة** التابعة ل **إيه سي إم** هي جيدة اليوم بمقدار ما كانت عندما كُتبت المراجعة:

ينبغي على كل مدير للمبرمجين أن يقتني نسخته الخاصة. ينبغي أن يقرأها، ويضعها في قلبه، ويعمل كما يملي عليه إحساسه، ويترك نسخة على مكتبه الشخصي كي تُسرق من قبل مبرمجيه. ينبغي أن يتابع إعادة وضع نسخ مكان المسروقة حتى يتم التوازن (ويس 1972).

إن لم تستطع إيجاد الكتاب، ابحث عن The *Mythical ManMonth*(Brooks 1995) أو *PeopleWare*(DeMarco and Lister 1999). كلاهما يحثان على موضوع أن البرمجة هي أولاً وفي المقام الأول أمر يقوم به الناس وفقط ثانوياً أمر صادف أن يحتوي الحواسيب.

نظرة عامة ممتازة وأخيرة عن قضايا تطوير البرمجيات هو *Software Creativity* (Glass 1995)، ينبغي أن يكون هذا الكتاب كتاباً خارقاً في الإبداع البرمجي بطريقة *Peopleware* التي كانت في فرق البرمجة. ناقش غلاس الإبداع مقابل الانضباط، والنظرية مقابل التطبيق، والاستدلالات مقابل المنهجية، والعملية مقابل المنتج، والعديد من الثنائيات التي تعرف مجال البرمجيات. بعد عدة سنوات من مناقشة هذا الكتاب مع مبرمجين عملوا لديّ، استخلصت أن صعوبة الكتاب كانت في أنه مجموعة مقالات مُعدلة من قبل غلاس لكنها ليست مكتوبة من قبله كلياً. لبعض القراء، هذا يعطي الكتاب شعوراً ناقصاً. مع هذا، لا أزال أطلب حتماً من كل مطور في شركتي أن يقرأه. الكتاب انتهت طباعته ومن الصعب إيجاده لكن يستحق الجهد إن كنت قادراً على إيجاده.

نظرات عامة على هندسة البرمجيات

ينبغي على كل مبرمجي الحاسوب الذين يتدربون ومهندسي البرمجيات أن يمتلكوا مراجعاً عالية المستوى في هندسة البرمجيات. تستعرض هكذا كتب المنظر الطبيعي المنهجي وليس رسماً لملامح خاصة بالتفصيل. إنها تؤمّن نظرة عامة على تطبيقات هندسة البرمجيات الفعالة وتغلف توصيفات تقنيات هندسة البرمجيات. توصيفات المُغلّف ليست مفصلة كفاية حتى تُمرِّن نفسك على التقنيات، لكن حتى يقوم كتاب واحد بهذا فيجب أن يكون عدة آلاف من الصفحات. إنها تؤمّن معلومات كافية حتى تستطيع تعلّم كيفية ملائمة التقنيات مع بعضها وتستطيع اختيار تقنيات لتحرٍ أبعد.

Roger S. Pressman's *Software Engineering: A Practitioner's Approach*, 6th ed. (Pressman 2004)، هو استعراض متوازن للمتطلبات، والتصميم، وتفعيل الجودة، والإدارة. تصرف صفحاته ال 900 انتباهاً قليلاً على التطبيقات البرمجية، لكن هذا قصور ثانوي، خصوصاً إن كنت مسبقاً تمتلك كتاباً عن البناء كهذا الذي تقرأ.

الإصدار السادس من Ian Sommerville's *Software Engineering*(Sommerville 2000) هو كتاب قابل للمقارنة مع كتاب بريسمان، وهو أيضاً يقدم نظرة عامة عالية المستوى جيدة عن عملية تطوير البرمجيات.

لائحة مراجع مشروحة أخرى

لوائح المراجع الحوسبية الجيدة نادرة.[1] إليك قلّة تسوّغ الجهد المبذول للحصول عليها:

ACM Computing Reviews ***مراجعات إيه سي إم الحوسبية*** هو منشور تابع ل **جمعية آليات الحوسبة (إيه سي إم)** والمكرّس لمراجعة كتب في كل جوانب الحواسيب وبرمجة الحواسيب. المراجعات منظَّمة حسب مخطط تصنيف شامل، ما يجعل إيجاد كتب في مساحة اهتمامك أمراً سهلاً. لمعلومات عن هذا المنشور وعن العضوية في **إيه سي إم**، انظر *www.acm.org*.

Construx Software's Professional Development Ladder (www.construx.com/ladder/). يؤمن موقع الوب هذا برامج قراءة لمطوري ومختبري ومدراء البرمجيات.[2]

35. 3 المجلات

مجلات المبرمجين غير المختصة

هذه المجلات غالباً متاحة في قوائم المجلات المحلية:

[1] cc2e.com/3502

[2] cc2e.com/3509

www.sdmagazine.com .Software Development "تطوير البرمجيات"[1] تركز هذه المجلة على قضايا البرمجة-أقل من النصائح بخصوص البيئات المحددة بالمقارنة مع القضايا العامة التي تواجهها كمبرمج محترف. جودة المقالات جيدة تماماً. إنها تتضمن أيضاً مراجعات للمنتجات.

www.ddj.com .Dr. Dobb's Journal هذه المجلة موجهة نحو المبرمجين المتعصبين للبرمجة[2]. تميل مقالاتها لتعالج قضايا تفصيلية وتتضمن الكثير من الشفرة.

إن لم تستطع إيجاد هاتين المجلتين في قوائم المجلات المحلية، سيرسِل لك العديد من الناشرين إصدارات مجانية، والعديد من المقالات متاح على الانترنت.

مجلات المبرمجين التخصصية

إنك لا تشتري هذه المجلات من قوائم المجلات بالعادة. عادة عليك أن تذهب إلى مكتبة جامعة مركزية أو تشترك بها لنفسك أو لشركتك:

www.computer.org/software .IEEE Software[3] تركز هذه المجلة نصف الشهرية على البناء والإدارة والمتطلبات والتصميم في البرمجيات بالإضافة إلى مواضيع برمجية أخرى في الحافة الأمامية أخرى. مهمتها هي أن "تبني مجتمعاً من ممارسي مهنة البرمجيات القياديين." في 1993، كتبتُ أنها كانت "المجلة الأكثر قيمة التي يمكن أن يشترك بها مبرمج." مذ أن كتبت ذلك، وأنا محرر أساسي في المجلة، ولا أزال أعتقد أنها أفضل مجلة متاحة لممارسي مهنة البرمجيات الجادّين.

www.computer.org/computer .IEEE Computer[4] هذه المجلة الشهرية هي المنشور **الأفضل لمجتمع الحاسوب في آي تربل إي (معهد مهندسي الكهرباء والإلكترون)**. إنها تنشر مقالات في طيف واسع من مواضع الحاسوب وتمتلك معايير مراجعة دقيقة لتؤكد على جودة المقالات التي تنشرها. بسبب اتساعها، ربما ستجد مقالات أقل تثير اهتمامك مما ستجد في *IEEE Software*.

www.acm.org/cacm ،Communications of the ACM[5] هذه المجلة واحدة من منشورات الحاسوب المتاحة الأقدم والأكثر احتراماً. إنها تمتلك امتيازاً واضحاً في النشر بالطول والعرض لعلوم الحاسوب، الموضوع

[1] cc2e.com/3516
[2] cc2e.com/3523
[3] cc2e.com/3530
[4] cc2e.com/3537
[5] cc2e.com/3544

الذي أصبح أفسح مما كان عليه حتى قبل عدة سنين. كما في IEEE Computer، بسبب اتساعها، ربما ستجد العديد من المقالات خارج مساحة اهتمامك. تميل المجلة إلى امتلاك نكهة أكاديمية، والتي لها جانب سيئ وجانب حسن. الجانب السيئ هو أن بعض الكتّاب يكتبون بأسلوب أكاديمي غامض. الجانب الحسن هو أنها تحتوي معلومات الحافة الأمامية التي لن تَرشَح نزولاً إلى المجلات غير التخصصية إلا بعد عدة سنوات.

منشورات لاهتمامات خاصة

تؤمن العديد من المنشورات تغطية عميقة لمواضيع متخصصة.

المنشورات الاحترافية

يصدر **مجتمع آي تربل إي الحاسوبي** صحفاً متخصصة بهندسة البرمجيات،[1] والأمن والخصوصية، والرسوم المتحركة ورسوميات الحاسوب، وتطوير الانترنت، ووسائل الإعلام، والأنظمة الذكية، وتاريخ الحوسبة، ومواضيع أخرى. انظر إلى www.computer.org لمزيد من التفاصيل.

تصدر **أيه سي إم** أيضاً منشورات لاهتمامات خاصة في الذكاء الاصطناعي،[2] والتفاعل بين الحواسيب والبشر، وقواعد البيانات، والأنظمة المضمنة، والرسوميات، ولغات البرمجة، وبرمجيات الرياضيات، والتشبيك، وهندسة البرمجيات، ومواضيع أخرى. انظر إلى www.acm.org لمزيد من التفاصيل

المنشورات المشهورة في السوق

هذه المجلات كلها تغطي ما توحيه أسماؤها[3]

مجلة مستخدمي السي / السي ++ www.cuj.com

مجلة مطوري جافا www.sys-con.com/java

برمجة الأنظمة المضمنة www.embedded.com

مجلة لينوكس www.linuxjournal.com

مراجعة يونكس www.unixreview.com

مطوري شبكات ويندوز www.wd-mag.com

[1] cc2e.com/3551

[2] cc2e.com/3558

[3] cc2e.com/3565

35.4 مخطط قراءة لمطور البرمجيات

يصف هذا القسم برنامج قراءة يلزم لمطور البرمجيات أن يعمل به حتى يحقق سمعة احترافية كاملة في شركتي، **كونستركس سوفتوير**.[1] المخطط الموصوف هو مخطط أساسي عام لممتهن البرمجيات الذي يريد أن يركز على التطوير. برنامجنا للقيادة الروحية مهيًا لتكييف إضافي للمخطط العام ليدعم اهتمامات الأفراد، وفي **كونستركس** هذه القراءة تُكمَّل بالتمرين والخبرات الاحترافية الموجهة.

المستوى الابتدائي

للانتقال إلى ما وراء المستوى الابتدائي في **كونستركس**، يجب أن يقرأ المطور الكتب التالية:

"دليل لأفكار أفضل" الإصدار الرابع

Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. Cambridge, MA: Perseus Publishing, 2001.

"درر البرمجة" الإصدار الثاني

Bentley, Jon. *Programming Pearls*, 2d ed. Reading, MA: Addison-Wesley, 2000.

Glass, Robert L. *Facts and Fallacies of Software Engineering*. Boston, MA: AddisonWesley, 2003.

"دليل استمرار مشروع البرمجية"

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998.

"الشفرة الكاملة" الإصدار الثاني

McConnell, Steve. *Code Complete*, 2d ed. Redmond, WA: Microsoft Press, 2004.

مستوى ممارس المهنة

لإتمام المقام "المتوسط" في **كونستركس**، يلزم للمبرمج أن يقرأ المواد الإضافية التالية:

" نماذج إدارة إعداد البرمجيات: العمل الجماعي الفعال، التكامل العملي "

Berczuk, Stephen P. and Brad Appleton. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston, MA: Addison-Wesley, 2003

[1] cc2e.com/3507

"دليل موجز للغة نمذجة الكائن القياسية " الإصدار الثالث

Fowler, Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3d ed. Boston, MA: Addison-Wesley, 2003

" إبداع البرمجيات "

Glass, Robert L. *Software Creativity.* Reading, MA: Addison-Wesley, 1995

"اختبار برمجيات الحاسوب" الإصدار الثاني

Kaner, Cem, Jack Falk, Hung Q. Nguyen*. Testing Computer Software*, 2d ed. New York, NY: John Wiley & Sons, 1999

" مقدمة إلى التحليل والتصميم الموجه بالكائنات والعملية الموحدة " الإصدار الثاني

Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 2001

"التطوير السريع"

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996

"متطلبات البرمجيات" الإصدار الثاني

Wiegers, Karl*. Software Requirements*, 2d ed. Redmond, WA: Microsoft Press, 2003

" كتيب المدير لتطوير البرمجيات "[1]

"Manager's Handbook for Software Development," NASA Goddard Space Flight Center.

يمكن تحميله من *sel.gsfc.nasa.gov/website/documents/online-doc.htm*

المستوى الاحترافي

يجب على مطور البرمجيات أن يقرأ المواد التالية ليحرز سمعة احترافية كاملة في **كونستركس** (مستوى "القيادة"). تُحاك متطلبات إضافية لكل مطور على حدة، يصف هذا القسم المتطلبات العامة.

[1] cc2e.com/3514

" هندسة البرمجيات في الممارسة " الإصدار الثاني

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, 2d ed Boston, MA: Addison-Wesley, 2003

"إعادة التصنيع: تحسين التصميم للشفرات الموجودة"

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999

"نماذج التصميم"

Gamma, Erich, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995

"مفاهيم إدارة هندسة البرمجيات"

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988

"كتابة شفرة متينة"

Maguire, Steve. *Writing Solid Code. Redmond*, WA: Microsoft Press, 1993

"بناء البرمجيات الموجهة بالكائنات" الإصدار الثاني

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997

"دليل قياس البرمجيات"[1]

“Software Measurement Guidebook,” NASA Goddard Space Flight Center. متاح من sel.gsfc.nasa.gov/website/documents/online-doc.htm

لمزيد من التفاصيل حول برنامج التطوير هذا، بالإضافة إلى لوائح قراءة مُحدَّثة، [2]شوف انظر إلى موقعنا للتطوير الاحترافي *www.construx.com/professionaldev/*.

[1] cc2e.com/3521

[2] cc2e.com/3528

35. 5 الانضمام إلى منظمات مهنية

واحدة من الطرق الأفضل لتعلم المزيد عن البرمجة هي أن تكون على تواصل مع المبرمجين الآخرين المتفانين في مهنتهم مثلك. [1]مجموعات المستخدمين المحلية الخاصة بمنتجات عتادية ولغوية هي إحدى أنواع المجموعات. الأنواع الأخرى هي المنظمات الاحترافية القومية والعالمية. المنظمة الأكثر توجهاً إلى الممارسة هي **مجتمع الحاسوب** في **إيه تربل إي**، والتي تصدر مجلتي *IEEE Computer* و *IEEE Software*، من أجل معلومات عن العضوية، انظر www.computer.org.

المنظمة الاحترافية الأولى كانت هي **إيه سي إم**، [2]والتي تصدر ""اتصالات ال إيه إس إم" *Communications of the ACM* والعديد من المجلات ذات الاهتمام الخاص. إنها تميل لتكون بطريقة ما موجهة أكثر للأسلوب الأكاديمي من **مجتمع الحاسوب** في **إيه تربل إي**. لمعلومات عن العضوية، انظر *www.acm.org*.

[1] cc2e.com/3535

[2] cc2e.com/3542

مسرد المصطلحات

المصطلح باللغة الإنكليزية	المصطلح العربي المُستخدم	أو
Architecture	هيكلة	
Argument	مُعامل	جِدال
Assertions	تأكيدات	مصادقات
Audit	تدقيق	
base class	صف القاعدة	الصف الاب
Bugs	ثغرات	
Centralized Exception Reporter	مُخبر الاستثناء المركزي	
Code	شفرة	تعليمات برمجية
Compiler	مترجم	
Comprehensive	الشمولية	
Configuration	تكوين	إعداد
Create	إنشاء	
Data	بيانات	معطيات
Debug	تصحيح	تنقيح
Defensive Programming	البرمجة الوقائية	
derived class	صف مشتق	الصف الابن
Construction	البناء	
design patterns	نماذج تصميم	
Enumerated	تعدادي	قابل للعدّ
Flag	إشارة	راية

تابع	دالة	Function
	البيانات الأساسية	Fundamental Data
عامة	شاملة	Global
معالجة، تعامل مع	تداوُل	Handling
	العتاد الصلب	Hardware
استكشافي (صفة)	استدلال (اسم)	Heuristic
	إجرائيات عالية الجودة	High-Quality Routines
دليل (شائع مع المصفوفات)	فهرس	Index
	معلومات	Information
	تفحص	Inspection
تنصيب	تثبيت	Installation
دمج	تكامل	Integrity
	تحري	Investigate
	التكرار	Iteration
التخطيط	التنسيق	Layout
	حلقات	Loops
ماكرو	مُسجّل	Macro
	منهجية	Methodology
طرق	مناهج	Methods
المتحولات الاسمية	المتغيرات الاسمية	Naming Variables
	خطأ "بعيداً من واحدة"	off-by-one error
	تنظيم الشفرة الخطيّة	Organizing Straight-Line Code
البرمجة بالأزواج	البرمجة الثنائية	pair programming

Parameters	محددات	وسطاء
polymorphic dispatch	الإرسال متعدد الأشكال	
Preprocessor	معالج تمهيدي	معالج تحضيري
primitive (data)	بدائي	أساسي
Procedure	اجراء	
Production Code	شفرة الإنتاج	الشفرة النهائية
Profile	موصّف	
Prototype	نسخة أولية	
Pseudocode	شفرة زائفة	
Recursion	العوديّة	
Referencing	تأشير	
Requirements	متطلبات	
Robustness	قوة	
Routine	إجرائية	روتين
Segment	مقطع	
Setup	إعداد	
Statement	تعليمة	عبارة
Stream	تدفق	
Structure	هيكل	بنية
Style	الأسلوب	النمط
Subscript	تدوين فرعي	
The Pseudocode Programming Process	عملية برمجة الشفرة الزائفة	
Upstream	تحضيري	

Upstream Prerequisites	المتطلبات الأولية التحضيرية	
Variable	متغير	متحول
Working Classes	الصفوف الناجحة	